MASTER EN SCIENCES INFORMATIQUES

Emulation de transactions réseaux au moyen de traces

Carouy, Etienne

Award date:
2005

Facultés Universitaires Notre-Dame de la Paix, Namur
Institut d'Informatique
Année Académique 2004 - 2005

Emulation de transactions réseaux au moyen de traces

Carouy Etienne

Mémoire présenté en vue de l'obtention du grade de Maître en Informatique.

Résumé

De nos jours, les systèmes distribués de communication sont d'usage courant car ils proposent des avantages indéniables en matière d'adaptabilité, de flexibilité, de robustesse et de performance. Néanmoins ce type d'architecture peut également être à l'origine de situations délicates, par exemple lors de la correction d'un bogue découvert dans une instance en production chez une société cliente. En effet, il n'est pas toujours aisé de reproduire dans un environnement de développement un cas d'utilisation rencontré en production. En outre, certaines sociétés clientes ont des spécificités (terminaux particuliers, serveurs propres) telles que le trafic réseau observé ne respecte pas intégralement les standards d'application. Lors de l'analyse a posteriori d'un incident, il est donc primordial de se rapprocher au mieux des conditions opérationnelles réelles afin de pouvoir reproduire, le plus fidèlement possible, le problème rencontré.

L'objet de ce mémoire est de présenter au lecteur un outil développé au cours du stage dans la société Nextenso. Celui-ci permet de rejouer sur les machines de développement des trames réseau capturées chez un client à l'aide d'Ethereal. Nous allons d'une part décrire les différentes contraintes que pose le rejeu pour ensuite présenter les solutions que nous avons proposées. Nous en décrirons les limitations, ce qui nous permettra, en finale, d'ouvrir de nouvelles pistes permettant de les lever.

Mots-clefs : Rejeu, Emulation, Trames, Fichiers de traces, Fichiers log.

Abstract

Nowadays, the distributed communication systems are frequently used because they offer many advantages such as adaptability, flexibility, robustness and performance. Nevertheless this type of architecture may also cause more intricate situations, for instance when fixing a bug discovered at a customer's site. Indeed, it is not always easy to reproduce in a development environment a use case encountered at a customer's site. Moreover, some customers have specificities (particular terminals, custom-made servers) such that their network traffic is not fully compliant with applicable standards. When fixing a bug, it is therefore fundamental to be able to mimic the operational conditions as close as possible in order to repeat the problem to be solved.

The purpose of this thesis is to present a tool developed during a traineeship at Nextenso. It enables the replay of network traffic captured at a customer's site, using Ethereal, within a development environment. We will first describe the various constraints set by the replay. We will then describe the solutions we proposed, and their limitations. We will end up opening new tracks enabling to lift these limitations up.

Keywords: Replay, Emulation, Frame, Trace file, Log file.

Avant-propos

Je tiens à remercier le Professeur Laurent Schumacher pour avoir accepté d'être le promoteur de ce mémoire, pour ses lectures attentives et son suivi lors de la réalisation de ce travail.

Je remercie également Monsieur Gilles Fleury, co-promoteur de ce mémoire, pour ses judicieux conseils et sa disponibilité durant la réalisation de mon stage. Je tiens également à remercier les différentes équipes de la société Nextenso, en particulier l'équipe Network Access, pour m'avoir accueilli dans leur bureau et m'avoir aidé durant les 5 mois de mon stage.

Je remercie ma famille pour m'avoir soutenu lors de ces études et lors la réalisation de ce mémoire

Je remercie enfin toutes les personnes qui, de près ou de loin, m'ont apporté une aide dans la rédaction de ce mémoire.

Finalement, je tiens à saluer tous mes amis de classe pour les moments inoubliables vécus durant ces trois années de maîtrise.

Table des matières

Table des figures

Glossaire

ASP : *Active Server Page*. Technologie Microsoft de création dynamique de pages Web, elle concurrence le CGI.

BSD : *Berkeley software distribution*. Licence libre de distribution de logiciel, une version modifiée de cette licence est compatible avec la GNU GPL, dont la particularité est qu'elle autorise toute personne à réutiliser le code comme bon lui semble et sans restriction. La version originale de la licence BSD permet à un logiciel propriétaire d'incorporer du code source libre contrairement à la licence GPL.

CGI : *Common Gateway Interface*. Programmes spécifiques situés sur un serveur qui peuvent être exécutés par un client à partir des pages HTML. Ils sont le plus souvent utilisés pour accéder à des bases de données.

GPL : *General Public License*. Elle a été écrite par Richard Stallman et Eben Moglen pour fixer les conditions légales de distribution des logiciels libres. Garanti à l'utilisateur les libertés d'exécuter le logiciel pour n'importe quel usage, d'étudier son fonctionnement et de l'adapter à ses besoins, de redistribuer des copies, d'améliorer le programme et de rendre publiques les modifications afin que l'ensemble de la communauté en bénéficie.

GPRS : *General Packet Radio Service.* Norme pour la téléphonie mobile dérivée du GSM permettant un débit de données plus élevé. On le qualifie souvent de 2,5G. Le G est l'abréviation de *génération* est le 2.5 indique que c'est une technologie à mi-chemin entre le GSM (2^{e} génération) et l'UMTS (3^{e} génération).

HSCSD *: High-Speed Circuit-Switched Data*. Développement du CSD (Circuit Switched Data), le mécanisme original de transmission des données sur le réseau mobile. Les différences proviennent de la capacité d'utiliser différentes méthodes de codage et plusieurs *time slot* afin d'augmenter la bande passante.

HTTP : *HyperText Transfert Protocol.* Protocole de communication réseau utilisé pour le transport de fichiers hyper-textes à travers l'Internet.

NIDS : N*etwork Intrusion Detection System.* Système permettant de détecter les activités malicieuses comme les dénis de service, le scannage de ports et les tentatives d'intrusion en surveillant le trafic réseau.

OSI : *Open Systems Interconnection*. Norme ISO 7498 créé dans le but d'offrir une base commune à la description de tout réseau informatique. Dans ce modèle, l'ensemble des protocoles d'un réseau est décomposé en 7 parties appelées *couches OSI*, numérotées de 1 à 7.

PHP : Acronyme récursif de PHP: Hypertext Preprocessor. Langage de script qui est principalement utilisé pour être exécuté sur un serveur Web, mais il peut fonctionner comme n'importe quel langage interprété en utilisant les scripts et son interpréteur sur un ordinateur. PHP permet de développer suivant le modèle procédural ou/et un modèle objet. On désigne parfois PHP comme une plate-forme plus qu'un simple langage.

SSL : *Secure Socket Layer.* Protocole de sécurisation des échanges sur Internet, développé à l'origine par Netscape. Il a été renommé en Transport Layer Security par l'IETF. Il fournit quatre objectifs de sécurité : l'authentification du serveur, la confidentialité des données échangées (ou session chiffrée), l'intégrité des données échangées et de manière optionnelle, l'authentification du client.

TCP : *Transmission Control Protocol.* Protocole de communication réseau permettant l'établissement d'une connexion entre 2 hôtes et l'échange de données. Il garanti la transmission des données ainsi que la réception de celles-ci dans le même ordre que leurs émissions.

TTL : *Time to live.* Champs de l'entête du protocole IP indiquant combien de sauts le paquets peut faire avant d'être ignorés.

UDP : *User Datagram Protocol.* Protocole de remise de paquets, simple, non-fiable, sans connexion, appartenant à la couche 4 du modèle OSI, détaillé dans la RFC 768.

UMTS : *Universal Mobile Telecommunications System*. Technologies de téléphonie mobile de troisième génération (3G). Elle est elle-même basée sur la technologie W-CDMA, standardisée par le 3GPP et constitue l'implémentation européenne des spécifications IMT-2000 de l'ITU pour les systèmes radio cellulaires 3G. L'UMTS est parfois aussi appelé 3GSM, soulignant l'interopérabilité qui a été assurée entre l'UMTS et le standard GSM auquel il succède.

WAP : *Wireless Application Service*. Protocole de communication réseau utilisé principalement pour accéder à l'Internet avec des appareils sans fil comme, par exemple, les téléphones mobiles.

WDP : *Wireless Datagram Protocol.* Base de la pile de protocoles WAP chargée de l'interface avec les protocoles de transmission de données utilisés par les opérateurs de télécoms

WML : *Wireless Markup Language*. Langage utilise pour spécifier le contenu et l'interface utilisateur pour les terminaux WAP, le WAP forum propose une DTD pour le WML.

Introduction

Malgré l'accroissement constant de la puissance des ordinateurs et étant donné que la complexité des traitements demandés s'amplifie dans le même temps, le recours à un système distribué est courant. Cette architecture particulière permet d'adapter à moindre coût les ressources matérielles en vue de fournir un service répondant à des critères spécifiques. Malheureusement, cette architecture augmente également la complexité de développement de par le fait que les traitements sont partagés sur plusieurs machines et qu'il devient alors plus difficile de cerner un mauvais fonctionnement.

Dans le cadre d'un stage au sein de la société *Nextenso SA,* nous avons été amené à développer un outil permettant de faciliter le développement et les tests de qualité d'une telle architecture en permettant de rejouer à partir de traces des scénarios d'utilisation. Nous allons donc expliquer le fonctionnement d'un outil permettant le rejeu de traces, essentiellement des traces réseaux capturées à l'aide du logiciel Ethereal.

Dés le début du stage, nous avons adopté le cas d'utilisation WAP de la *Proxy Platform* parce qu'il présentait plusieurs aspects intéressants. Premièrement, ce cas de figure emploie à la fois les deux protocoles les plus utilisés comme protocole de communication c'est – à – dire UDP/IP et TCP/IP. Deuxièmement, il représentait une interaction complète avec la *Proxy Platform* dans le sens où il faisait intervenir plusieurs Stacks et Agents simultanément. Enfin, le support du protocole WAP étant implémenté depuis longtemps dans la *Proxy Platform*, le comportement de la *Proxy Platform* étant dès lors bien connu.

Le premier chapitre permettra aux lecteurs de se familiariser avec l'environnement particulier dans lequel le travail exposé dans ces pages a été développé. Nous commencerons par introduire le lecteur à la *Nextenso Proxy Platform* qui est la plateforme spécifique développée par la société *Nextenso SA* dans lequel les solutions ont été implémentées. Ensuite, nous poserons les bases nécessaires concernant le protocole WAP afin d'assurer une bonne compréhension de la problématique et de ses solutions.

Dans le second chapitre, nous définirons la notion de traces, en particulier les traces capturées par le logiciel Ethereal. Par la suite, nous présenterons les différents scénarios d'utilisation dans lequel le rejeu offre des avantages. Nous continuerons ce chapitre par décrire les mécanismes nécessaires qui ont été développé afin de capturer facilement des traces dans la *Proxy Platform*. Enfin et pour terminer le chapitre, nous définirons les spécifications du rejeu de traces.

Le troisième chapitre sera consacré à la description des solutions développées au cours du stage dans la société *Nextenso SA*. Nous introduirons ce chapitre en présentant un logiciel, en l'occurrence tcpreplay, sur lequel les solutions se basent, ainsi qu'un bref rappel concernant la composition d'une trame. Nous aborderons alors l'émulation client, en présentant les différentes contraintes qu'il faudra respecter, afin de permettre le rejeu d'un terminal client à partir de traces suivi par la description de la solution spécifique développée. Nous passerons dès lors à l'émulation serveur en énumérant les différentes contraintes propres pour finir par décrire la solution développée. La dernière section de ce chapitre sera consacrée à l'explication des limitations rencontrées dans les deux solutions précédemment décrites.

Le dernier chapitre développera trois pistes futures qui pourront être suivies afin de remédier aux limitations décrites. Ce chapitre sera également l'occasion de critiquer les outils existants et de montrer leurs propres limites.

1. La Proxy PlatForm

1.1 Introduction

Ce chapitre sera consacré à la présentation de la *Nextenso Proxy Platform,* qui est la plateforme de développement sur laquelle les travaux exposés dans ce document ont été menés.

Elle sera en premier lieu présentée de manière générale, puis on s'attardera davantage sur l'environnement dans lequel l'objet de ce mémoire a été développé.

Ce chapitre permet donc de situer le cadre de travail et permettra de mieux comprendre les solutions apportées.

1.2 Architecture générale

La *Nextenso Proxy Platform* est une plate-forme de développement facilitant la réalisation de solutions logicielles orientées réseau. Elle s'intègre dans un flux réseau quelconque à la manière d'un proxy et permet d'agir sur celui-ci.

Elle est flexible et permet d'écrire très facilement des logiciels respectant ce critère de qualité de plus en plus important. A cette fin, chaque application écrite pour la *Proxy Platform* peut être instanciée sur une ou plusieurs machines. Ces instances sont indépendantes et peuvent être lancées ou arrêtées de manière à s'adapter à la capacité de traitement demandé. Elle est également administrée via une interface *web* [Marquet].

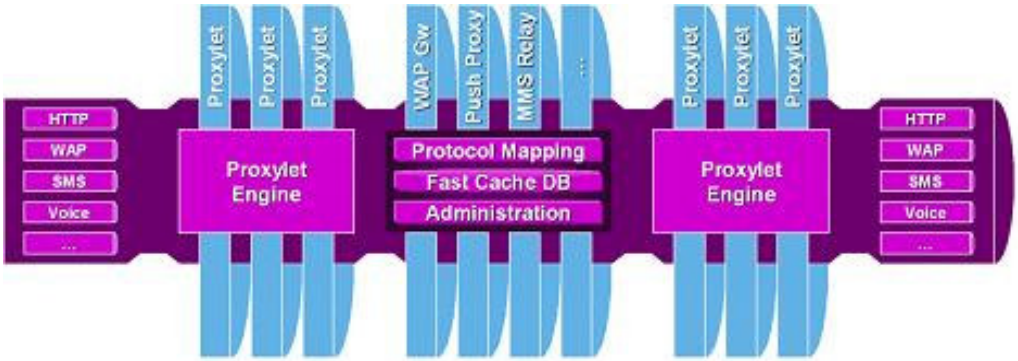

FIG 1.1 - Illustration de la Proxy Platform

La *Proxy Platform* abrite deux types de composantes principales : les *Stacks* et les C*allout Agents* dont les rôles sont complémentaires.

Le but de la stack est de gérer les connexions avec les composants extérieurs à la *Proxy Platform* mais également avec les autres stacks implémentées dans celle-ci. En effet chaque stack est dédiée à un protocole tel que l'HTTP, le WAP (*Wireless Application Protocol)*, …. A cette fin, elles sont écrites en C afin de répondre à des critères évidents de performance.

Le Callout Agent ou Agent quant à lui permet d'intervenir sur le contenu du flux intercepté par la Stack. Un agent est en fait un ensemble de mini applications écrites en Java : les *Proxylets*. Cette décomposition en mini applications apporte principalement deux avantages, elle permet de facilement augmenter la capacité de traitement de la *Proxy Platform* en multipliant les *Proxylets* mais également de décomposer des opérations très complexes en découpant logiquement le traitement à effectuer en opérations élémentaires. De plus, cette décomposition permet également de modifier facilement le traitement, d'ajouter certaines fonctionnalités, d'en retirer d'autres suivant les besoins du client. Ce concept permet à la *Proxy Platform* d'être modulable tout en facilitant grandement les phases d'implémentation et de maintenance.

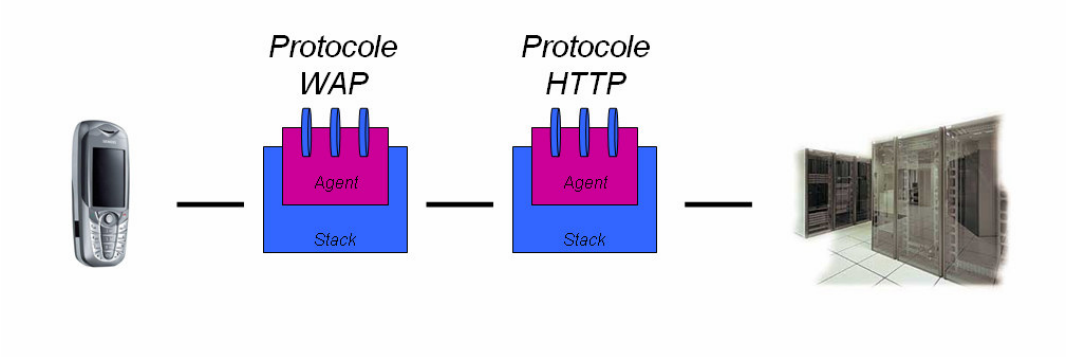

FIG 1.2 - Illustration de l'enchaînement des couples WAP et HTTP

Afin de traiter un protocole particulier, il est néanmoins nécessaire d'implémenter les 2 types de composants car ils sont utilisés en couple. En effet, les rôles sont complémentaires. Comme nous pouvons le voir sur la figure 1.2, ces couples peuvent être chaînés afin de faire interagir plusieurs protocoles. Dans le cas qui nous intéresse, le couple dédié au protocole WAP est associé au couple dédié au protocole HTTP.

Comme nous venons de le dire, une *proxylet* n'implémente jamais la totalité de la solution. Chacune va réaliser une partie de la solution et le chaînage permettra de réaliser la totalité du traitement. Remarquons également que les *proxylets* elles – mêmes sont chaînées sur deux flux de données : celui de la requête et celui de la réponse. Cette séparation permet une plus grande modularité comme nous pouvons le voir sur la figure 1.3.

Les chaînes de *proxylets* sont rassemblées dans des contextes qui peuvent ensuite être assignés à une instance d'agent. Néanmoins, il faut savoir que chaque contexte ne peut contenir qu'une seule instance de chaque *proxylet* par flux.

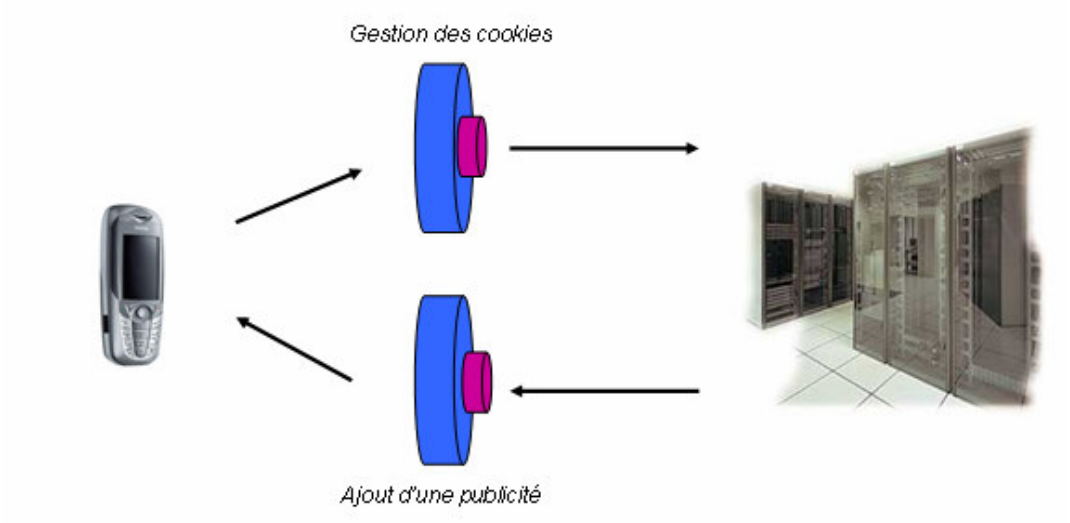

FIG 1.3 - Exemple du double flux des proxylets

Nous allons maintenant décrire plus précisément l'environnement dans lequel l'objet de ce mémoire a été réalisé c'est-à-dire le WAP.

1.3 WAP et rôle de la passerelle WAP

Le WAP propose de définir la façon par laquelle les terminaux mobiles accèdent à des services Internet, et cela à un niveau au-dessus de la transmission des données. Le WAP définit aussi la manière dont doivent être structurés les documents, grâce à un langage dérivant du HTML et nommé pour l'occasion WML (*Wireless Markup Language*) et un langage de script baptisé *WMLScript*. Etant donné les restrictions engendrées par le réseau d'accès (bande passante réduite) et le terminal (écran réduit, mémoire en petite quantité, de faibles capacités en termes de processeur, autonomie restreinte), il était nécessaire de mettre au point un protocole spécifique [Pillou].

Le terminal mobile (un appareil mobile tel qu'un téléphone supportant les fonctionnalités du WAP, un assistant personnel, ou bien tout autre appareil capable de supporter cette technologie) désirant obtenir des données en provenance d'un service WAP doit dans un premier temps se connecter à une passerelle à l'aide d'un numéro de téléphone, ou bien un assistant de connexion qui le composera pour l'utilisateur (de la même façon que pour accéder à Internet par modem par l'intermédiaire d'un fournisseur d'accès). Lorsque le terminal mobile

est connecté à la passerelle, l'ensemble des transactions effectuées par le mobile sont envoyées par la passerelle au serveur applicatif par une transmission de type IP, sous forme de requêtes proches du standard HTTP.

Le serveur applicatif va donc renvoyer à la passerelle des documents au format WML en fonction des requêtes du terminal mobile. Cela signifie que le serveur peut utiliser les mêmes technologies qu'un serveur web pour fournir ses données (accès à une base de données, exécution d'un script CGI, exécution de scripts PHP ou ASP, ou bien de servlets, …). Par conséquent, les applications possibles de cette technologie sont très vastes.

Une fois les données formatées, celles-ci sont envoyées à la passerelle, qui va se charger de les transmettre au terminal mobile par l'intermédiaire du réseau cellulaire. La passerelle a dans un premier temps un rôle d'interface entre le mobile fonctionnant sur un réseau cellulaire, et le réseau IP, fonctionnant sur un support quelconque. Toutefois le rôle de la passerelle ne s'arrête pas là. En effet, celle-ci permet de transformer les réponses en provenance du serveur applicatif en données binaires compactées, donc beaucoup plus adaptées à transiter sur le réseau cellulaire de bande passante plus faible. Lorsque le terminal mobile reçoit ces données compressées, il les décode à l'aide d'un circuit électronique prévu à cet effet.

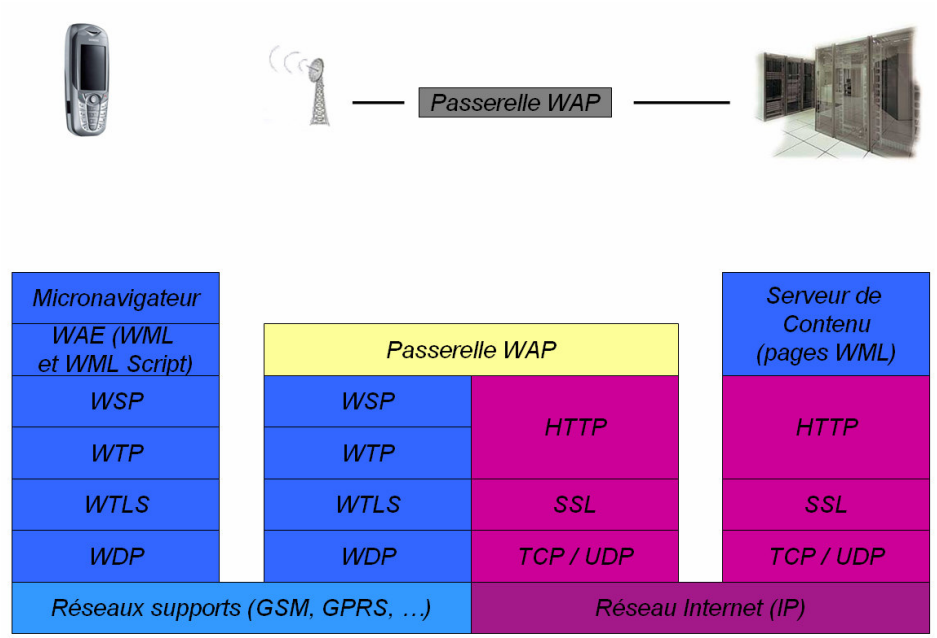

FIG 1.4 - Illustration de la passerelle WAP

Le protocole WAP est défini selon des couches, c'est-à-dire des niveaux d'abstraction de données (dans l'esprit du modèle OSI) afin de séparer les différents traitements des données nécessaires pour effectuer la transaction. Le protocole WAP est scindé en cinq couches :

- La couche WAE (Wireless Application Environment)

  La couche application du WAP définit l'environnement de développement des applications sur les terminaux mobiles. Elle fournit ainsi des fonctionnalités applicatives telles que le WML et le WMLScript déjà citées , mais également le WTA (*Wireless Telephony Applications*) qui est un ensemble d'interfaces prédéfinies servant à créer des applications téléphoniques

- La couche WSP (Wireless Session Protocol)

  La couche session du WAP est constituée de deux protocoles : un protocole orienté connexion agissant au-dessus de la couche transaction et un protocole non orienté connexion agissant au-dessus de la couche transport. La présence de ces deux protocoles permet de bénéficier soit de longues sessions sans acquittement, dans lesquelles la communication peut être suspendue puis reprise, ou bien de sessions initiées par le serveur (technologie *PUSH*)

- La couche WTP (Wireless Transaction Protocol)

  La couche de transaction gère le déroulement de la transaction, elle définit donc la fiabilité du service. La communication peut se faire de trois façons, à sens unique avec acquittement, à sens unique sans acquittement, en full duplex avec acquittement. Elle permet en outre d'effectuer des transactions synchrones et de retarder les acquittements afin de les gérer par paquets.

- La couche WTLS (Wireless Transport Layer Security)

  Puisque les données circulent entre le terminal mobile et la passerelle grâce à des réseaux sans fil, il est nécessaire que les transactions soient sécurisées, c'est ce que la couche sécurité se propose de faire. Celle-ci est basée sur le standard *SSL* (*Secure Socket Layer*) et permet de crypter les échanges de données, de

garantir l'intégrité des données (vérifier que celles-ci n'ont pas été modifiées) et d'authentifier les acteurs de l'échange.

- La couche WDP (Wireless Datagram Protocol)

  La couche WDP est à la base de la pile de protocoles WAP, c'est elle qui est chargée de l'interface avec les protocoles de transmission de données utilisées par les opérateurs de télécoms telle que HSCSD, GPRS, UMTS, …

Pour revenir à la *Proxy Platform* et comme dit précédemment, il est nécessaire d'implémenter deux types de composants afin d'agir sur le flux WAP, et par analogie avec le flux HTTP généré par celui-ci comme représenté par la figure 1.5.

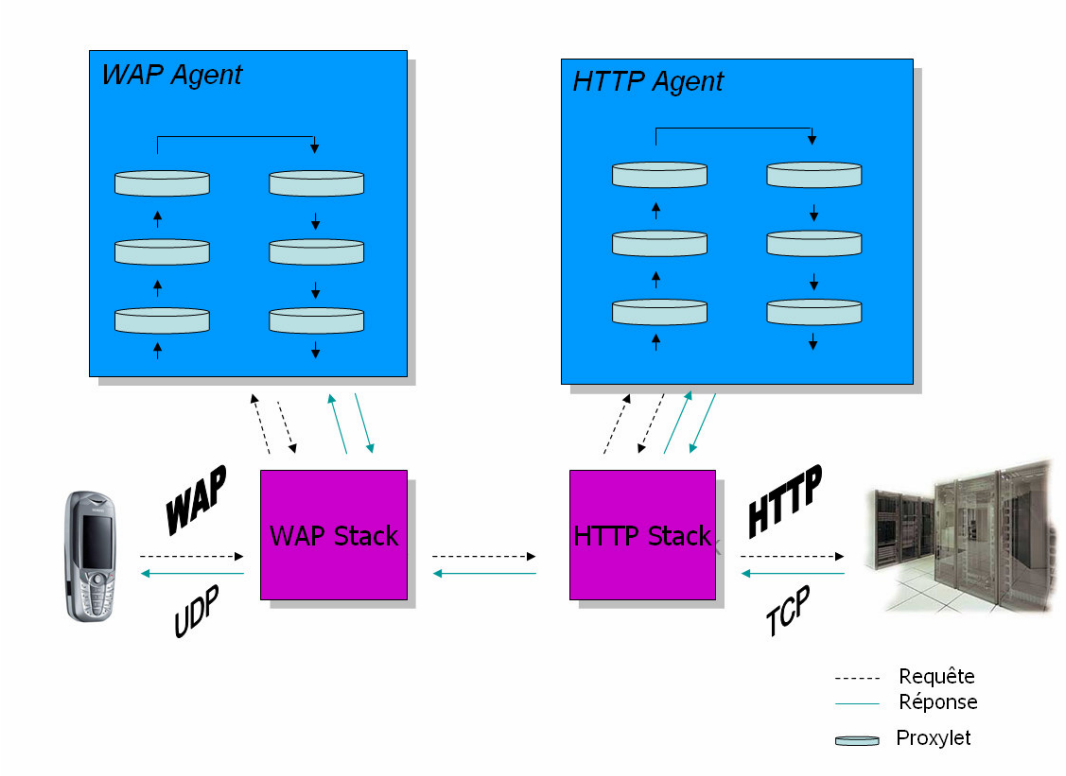

FIG 1.5 - Illustration du traitement d'une requête et de sa réponse pour le protocole WAP

On peut constater que le couple *WAP Stack – WAP Agent* remplit une partie du rôle de la passerelle WAP définit plus haut en adaptant le contenu des trames. Les proxylets contenues dans l'agent permettent de modifier le contenu des requêtes en changeant par exemple l'encodage des images et leurs caractéristiques (hauteur, largeur) afin d'être affichées de manière optimale par le terminal mobile (les caractéristiques techniques du terminal sont

connues car contenues dans la première requête envoyée par celui-ci). Remarquons enfin que pour le rejeu, nous allons utiliser le protocole WAP/UDP et le protocole HTTP/TCP.

1.4 Exemple d'utilisation du WAP

Nous allons détailler dans cette section une interaction WAP – HTTP afin de comprendre le rôle des différents paquets transmis et leur relation, et ce dans le but de permettre une meilleure compréhension de la problématique décrite dans le chapitre suivant de ce mémoire. Nous allons donc prendre un exemple concret.

Avant toute chose, lorsque un utilisateur désire utiliser des services Internet à partir de son mobile, il doit dans un premier temps obtenir une adresse IP qui lui permettra de se connecter à une passerelle WAP. Cette adresse IP est obtenue par l'émission d'une requête RADIUS.

Voici les traces prises par Ethereal [1]d'une brève interaction entre un terminal et la Proxy Platform, suivies d'une explication du rôle de chacune des trames.

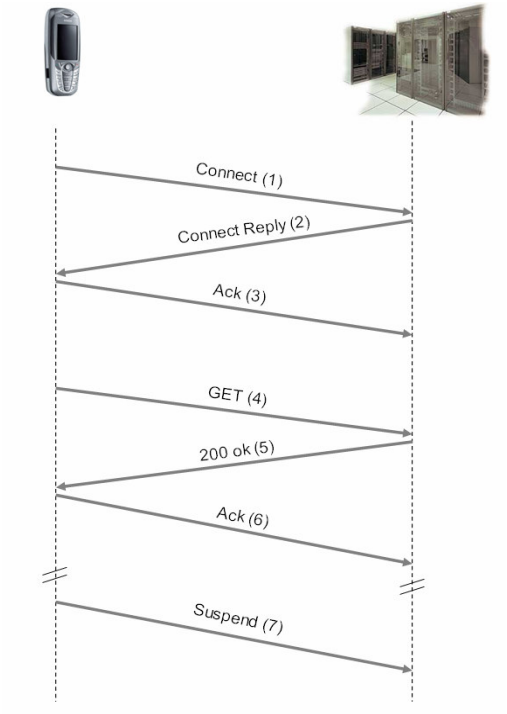

FIG 1.6 – Exemple de transaction WAP [Kurose&Ross]

[1] Le logiciel ethereal est décrit au chapitre suivant.

1) 172.20.3.162 -> 139.54.130.139 WTP+WSP WSP Connect (0x01)
2) 139.54.130.139 -> 172.20.3.162 WTP+WSP WSP ConnectReply (0x02)
3) 172.20.3.162 -> 139.54.130.139 WTP+WSP WTP Ack
4) 172.20.3.162 -> 139.54.130.139 WTP+WSP WSP Get (0x40) http://nx0097.nextenso.alcatel.fr:7777/llv.wml
5) 139.54.130.139 -> 172.20.3.162 WTP+WSP WSP Reply (0x04): 200 OK (0x20)
6) 172.20.3.162 -> 139.54.130.139 WTP+WSP WTP Ack
7) 172.20.3.162 -> 139.54.130.139 WTP+WSP WSP Suspend (0x08)

Une fois une adresse IP obtenue, le mobile doit se connecter à la passerelle WAP en utilisant une méthode similaire au « *hand shaking* » du TCP. Pour rappel, cela consiste en une identification décomposée en 3 étapes :

- Le mobile qui est l'initiateur de la connexion envoie une première requête de demande de connexion (Trame 1). Cette trame contient un nombre important de données notamment les caractéristiques propres au terminal. Voici un exemple de contenu d'une trame de connexion (décodée à l'aide d'Ethereal):

Encoding-version: 1.3
Accept: application/vnd.wap.wmlc
Accept: application/vnd.wap.wmlscriptc
Accept: application/vnd.wap.multipart.related
Accept: application/vnd.wap.multipart.mixed
Accept: application/vnd.phonecom.mmc-wbxml
Accept: application/octet-stream
Accept: application/vnd.openwave.pp
Accept: text/plain
Accept: text/css
Accept: image/bmp
Accept: image/gif
Accept: image/jpeg
Accept: image/png
Accept: image/vnd.wap.wbmp
Accept: image/x-up-wpng
Accept: application/vnd.wap.sic
Accept: application/vnd.wap.slc
Accept: application/vnd.wap.coc
Accept: audio/midi
Accept: audio/amr
Accept: audio/x-sagem1.0
Accept: audio/x-sagem2.0
Accept: image/wbmp
Accept: image/vnd.wap.wbmp
Accept: image/bmp
Accept: image/png
Accept: image/jpeg
Accept: image/gif
Accept: text/x-vCard
Accept: text/x-vCalendar
Accept: application/vnd.uplanet.bearer-choice-wbxml
Switching to WSP header code-page: 1
User-Agent: SAGEM-myX-6/1.0 UP.Browser/6.1.0.6.1.c.4 (GUI) MMP/1.0
Accept-Charset: utf-8

Accept: application/smil
Accept: application/vnd.wap.mms-message
Accept: audio/wav
Accept: audio/x-wav
Accept: audio/iMelody
Accept: text/x-iMelody

Accept-Language: fr-fr
Profile: http://extranet.sagem.com/UAProfile/823622.xml
x-up-devcap-iscolor: 1
x-up-devcap-max-pdu: 100000
x-up-devcap-numsoftkeys: 2
x-up-devcap-screendepth: 65536
x-up-devcap-screenpixels: 128, 160
x-up-devcap-softkeysize: 5

Nous pouvons constater que le nombre d'informations contenues dans cette trame est important et que le caractère de ces informations est très varié. Néanmoins, c'est grâce à toutes ces informations que l'agent WAP va pouvoir adapter le contenu des réponses provenant d'Internet. Un exemple simple est le changement de la profondeur de couleur des images contenues dans la réponse afin de l'adapter et de permettre un affichage optimal sur le mobile.

- Ensuite, la WAP Gateway accepte cette connexion en envoyant une réponse endéans un temps prédéterminé. (Trame 2)

- Le mobile confirme l'établissement de la connexion. (Trame 3)

A partir de ce moment, le mobile est identifié et connecté. Il peut accéder aux services désirés, en envoyant des requêtes spécifiques. Dans notre exemple, nous pouvons constater que l'utilisateur a demandé l'affichage de la page *http://nx0097.nextenso.alcatel.fr:7777/llv.wml*. (Trame 4) que le serveur a renvoyé (Trame 5, acquittement du client : Trame 6). L'utilisateur n'ayant plus émis de requête dans un laps de temps prédéterminé, le mobile a automatiquement envoyé une trame *Suspend* (Trame 7) afin de libérer les ressources utilisées.

Comme on peut le constater, le mode de fonctionnement est fort similaire au TCP et l'accès à des pages Internet, bien que ces pages soient spécifiques, est également comparable.

2. Description de la problématique

2.1 Introduction

Ce chapitre va décrire l'objectif du stage mené au sein de la société Nextenso pendant les premiers mois de cette année académique. Il va donc introduire la problématique à laquelle ce mémoire propose des solutions dans les chapitres suivants, à savoir l'émulation de transactions réseaux à partir de traces.

Dans un premier temps, nous allons introduire la notion de traces et en particulier, les traces obtenues grâce à l'analyseur réseau Ethereal. Nous expliquerons ensuite de manière très brève les fichiers logs.

Ensuite, ce chapitre décrira les objectifs et les enjeux du rejeu de traces Ethereal, et d'une manière générale, les avantages consécutifs à l'émulation de transactions réseaux.

Pour finir, ce chapitre établira les spécifications du rejeu de traces Ethereal et les principaux critères constituant le cahier des charges.

2.2 Qu'est-ce qu'une trace

Une trace est un fichier contenant un relevé des activités d'une machine. Ces relevés peuvent prendre plusieurs formes : il peut s'agir de cookies qui permettent à un navigateur Internet de modifier le contenu de sa page en fonction des préférences de l'utilisateur lors de visites précédentes, un système de journalisation des événements, ou bien dans le cas qui nous intéresse un relevé de l'activité réseau de la machine pendant un certain temps.

2.2.1. Introduction au logiciel Ethereal

Ethereal est un analyseur de réseau libre sous licence GPL, fonctionnant à la fois sous Unix et Windows. C'est en 1997 que le projet Ethereal est lancé par Gerald Combs, qui depuis a été rejoint par une grande communauté de développeurs.

Ses principales fonctionnalités sont :

- ✓ la capture sur le réseau des paquets via la bibliothèque *pcap* sous Unix et *Winpcap* sous Windows,
- ✓ la lecture de nombreux formats de traces, générés par d'autres analyseurs réseau,
- ✓ le décodage de nombreux protocoles de différents niveaux via des dissecteurs dédiés,
- ✓ un système de filtrage des paquets afin de ne sélectionner que les paquets qui nous intéressent,
- ✓ une interface graphique simple et efficace divisée en 3 panneaux : résumé des trames (1), détail d'une trame décodée (2), données brutes décodées (3).

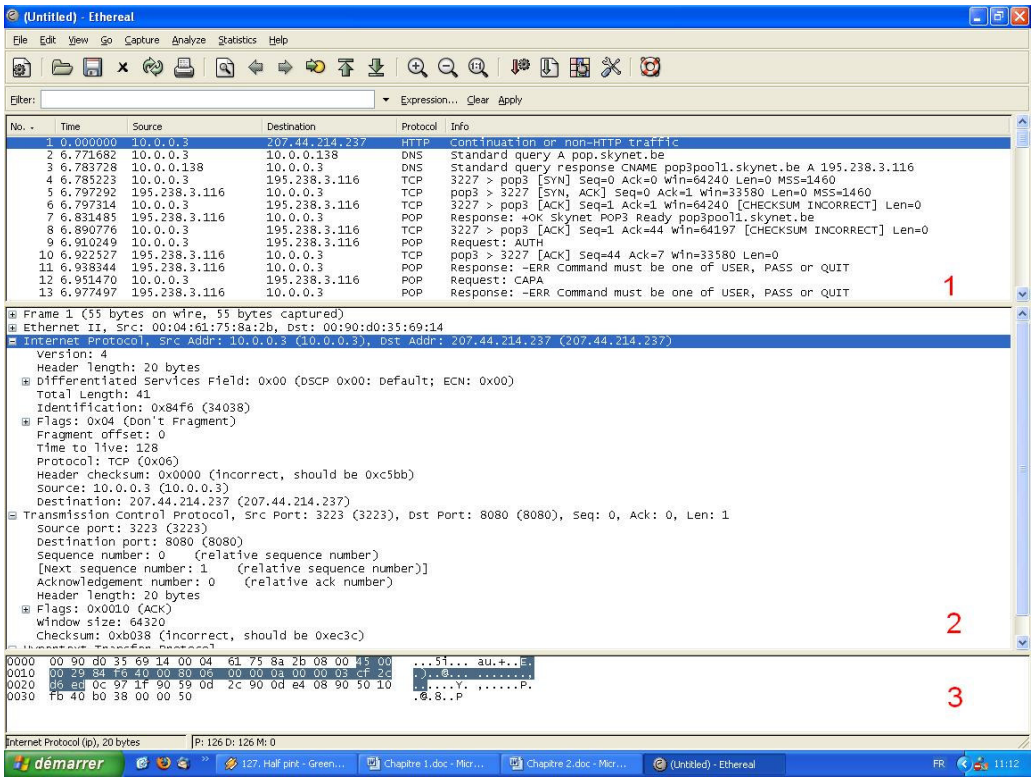

FIG 2.1 – Illustration de l'interface d'Ethereal

2.2.2. Introduction aux traces d'Ethereal

Comme nous venons de le dire, Ethereal permet la dissection d'un grand nombre de protocoles de différents niveaux. Afin de se familiariser quelque peu avec Ethereal, qui est devenu un logiciel incontournable lors de la réalisation d'applications réseaux, mais aussi de définir plus précisément la notion de traces nous allons en étudier une en détail.

Dans la figure 2.1, la partie supérieure de la fenêtre contient un bref descriptif des trames qui viennent d'être capturées. Chaque trame est résumée en une ligne décomposée en colonnes : les premières décrivent les informations générales telles que l'instant d'arrivée, l'adresse IP source, l'adresse IP destinataire, le protocole de plus haut niveau identifié ; la dernière colonne décrit quant à elle la trame et les données qu'elle contient de manière succincte. Voici un exemple de résumé d'une trame.

Time	Source	Destination	Protocol	Info
4.296188	10.0.0.3	66.102.9.104	HTTP	GET /firefox?client=firefox-a&rls=org.mozilla:fr-FR:official HTTP/1.1

Comme on peut le voir, il s'agit donc d'une trame HTTP (plus particulièrement d'une requête GET) provenant de la machine ayant l'adresse IP 10.0.0.3, à destination de la machine ayant l'adresse IP 66.102.9.104.

Ensuite, dans la deuxième fenêtre, on peut prendre connaissance de manière plus détaillée du contenu de la trame. En effet, grâce à un système de dissecteurs dédiés, Ethereal est capable de décoder tous les protocoles utilisés dans la trame et d'afficher toutes les données pour chacun d'eux. Malgré un nombre important d'informations, la lecture de celles-ci reste aisée grâce à un système hiérarchique qui permet de consulter le contenu protocole par protocole. Voici le détail de la trame correspondant à l'exemple utilisé pour le résumé :

Frame 1 (644 bytes on wire, 644 bytes captured)
Arrival Time: Mar 29, 2005 11:58:40.919635000
Protocols in frame: eth:ip:tcp:http

Ethernet II, Src: 00:04:61:75:8a:2b, Dst: 00:90:d0:35:69:14
Destination: 00:90:d0:35:69:14 (ThomsonB_35:69:14)
Source: 00:04:61:75:8a:2b (EpoxComp_75:8a:2b)
Type: IP (0x0800) **ETHERNET 802.3**

Internet Protocol, Src Addr: 10.0.0.3 (10.0.0.3), Dst Addr: 216.239.59.104 (216.239.59.104)
Version: 4
Header length: 20 bytes
Differentiated Services Field: 0x00 (DSCP 0x00: Default; ECN: 0x00)
0000 00.. = Differentiated Services Codepoint: Default (0x00)
.... ..0. = ECN-Capable Transport (ECT): 0
....0 = ECN-CE: 0
Total Length: 630
Identification: 0x8a0d (35341)
Flags: 0x04 (Don't Fragment)
0... = Reserved bit: Not set
.1.. = Don't fragment: Set
..0. = More fragments: Not set
Fragment offset: 0
Time to live: 128
Protocol: TCP (0x06)
Header checksum: 0x501a
Source: 10.0.0.3 (10.0.0.3)
Destination: 216.239.59.104 (216.239.59.104) **IP V4**

Transmission Control Protocol, Src Port: 3274 (3274), Dst Port: http (80), Seq: 0, Ack: 0, Len: 590
Source port: 3274 (3274)
Destination port: http (80)
Sequence number: 0 (relative sequence number)
Next sequence number: 590 (relative sequence number)
Acknowledgement number: 0 (relative ack number)
Header length: 20 bytes
Flags: 0x0018 (PSH, ACK)
0... .... = Congestion Window Reduced (CWR): Not set
.0.. .... = ECN-Echo: Not set
..0. .... = Urgent: Not set
...1 .... = Acknowledgment: Set
.... 1... = Push: Set
.... .0.. = Reset: Not set
.... ..0. = Syn: Not set
....0 = Fin: Not set
Window size: 63605
Checksum: 0x20c3 **TCP**

Hypertext Transfer Protocol **HTTP**
GET /firefox?client=firefox-a&rls=org.mozilla:fr-FR:official HTTP/1.1\r\n
Request Method: GET
Request URI: /firefox?client=firefox-a&rls=org.mozilla:fr-FR:official
Request Version: HTTP/1.1
Host: www.google.be\r\n
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; fr-FR; rv:1.7.6) Gecko/20050318 Firefox/1.0.2\r\n
Accept: ext/xml, pplication/xml,application/xhtml+xml, ext/html;q=0.9,text/plain;q=0.8, mage/png,*/*; =0.5\r\n
Accept-Language: fr,fr-fr;q=0.8,en-us;q=0.5,en;q=0.3\r\n
Accept-Encoding: gzip,deflate\r\n Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7\r\n
Keep-Alive: 300\r\n
Connection: keep-alive\r\n
Cookie: PREF=ID=ab657d86ec6d76dd:LD=fr:TM=1111682496:LM=1111694518:S=Zay_Mo95iaUF_9ID\r\n
Cache-Control: max-age=0\r\n\r\n

Enfin et afin d'être complet, Ethereal affiche dans sa dernière fenêtre le dump complet de la trame en hexadécimal, qui permet de vérifier et/ou de saisir comment les données sont réellement transmises. A cette fin, un système de surlignement permet à l'utilisateur de facilement repérer les bytes concernés par le protocole qu'il est en train d'analyser dans la deuxième fenêtre.

2.2.3. Fichier log

Un fichier log est un fichier regroupant l'ensemble des événements survenus sur un logiciel, une application, un serveur ou tout autre système informatique.

Un fichier log se présente sous la forme d'un fichier texte classique, reprenant de façon chronologique, l'ensemble des événements qui ont affecté un système informatique et l'ensemble des actions qui ont résulté de ces événements. Ainsi, par exemple et pour un serveur de type Web, le fichier log pourrait regrouper pour chaque demande d'accès à chacun des fichiers du serveur :

- ✓ la date et l'heure précise de la tentative d'accès ;
- ✓ l'adresse IP du client ayant réalisé cet accès ;
- ✓ le fichier cible ;
- ✓ le système d'exploitation et le navigateur utilisé pour cet accès ;
- ✓ la réponse fournie par le serveur à cette demande d'accès.

Le fichier log peut être utilisé pour diverses raisons, par exemple la journalisation de tous les événements d'un serveur web qui permettront ensuite d'établir diverses statistiques. Ces fichiers sont également souvent utilisés par les développeurs pour le débogage afin de pouvoir suivre l'évolution de l'application dans les instants qui ont précédé son dysfonctionnement. Ils sont malheureusement difficilement exploitables dans des applications multi machines car cela nécessite de regrouper les fichiers log des différentes machines et de les recouper afin de bien comprendre ce qui s'est passé avant le dysfonctionnement.

2.3 Scénarios d'utilisation du rejeu

L'objectif de ce mémoire est donc l'émulation de transactions réseaux par le rejeu de traces. Afin de montrer les avantages de la possibilité de rejouer des traces, et de saisir ce que cela implique dans les différentes activités liées au développement de logiciels réseaux, nous allons détailler quelques cas d'utilisation : l'amélioration des tests de validation, le reproduction de bogue et l'optimisation des paramètres.

2.3.1. Amélioration des tests de validation.

Aujourd'hui, l'équipe de validation est dans l'obligation de tester les composants de la Proxy Platform en suivant des tests rigoureux qui ont été établis à l'avance. Ces tests doivent être renouvelés à chaque nouvelle version de la Proxy Platform.

Ces tests portent sur les performances, l'absence de dysfonctionnements, la compatibilité entre la Proxy Platform et différents modèles de mobiles,…

On peut donc constater que le domaine de test est assez large. Surtout que l'équipe est obligée d'utiliser plusieurs mobiles pour chaque cas de figure car ceux-ci ne se comportent pas toujours de manière identique.

L'apport d'un système de rejeu est ici évident. En effet, les tests définis par l'équipe de validation sont enregistrés sous forme de traces et rejoués, ce qui permet un gain de temps et une uniformisation des tests. De plus, cela permet d'alléger quelque peu le processus de validation en automatisant certains des tests à effectuer. En effet pour chaque bogue identifié, l'équipe de validation rédige à l'heure actuelle un ticket adressé aux développeurs chargés du composant qui vient d'être mis en cause. Une fois le ticket émis, ces développeurs sont invités à corriger le problème. Ensuite l'équipe de validation devra refaire le test avant de pouvoir classer le ticket. On constate donc que la correction d'un bogue est un processus qui peut prendre du temps, même pour un bogue de faible importance. Un système de rejeu permettrait à l'équipe de développeurs de facilement valider leur correction.

La possibilité d'exécuter des tests automatiques plus complexes que ceux permis par des JUnit est également envisageable avec un système de rejeu. En effet, l'utilité d'une JUnit est de

pouvoir tester une classe en particulier. L'apport d'un système de rejeu permettrait de tester les interactions entre les classes.

Chaque nuit, toutes les sources sont recompilées afin de prendre en compte les dernières modifications effectuées pendant la journée. Nous pouvons donc imaginer que des tests automatiques permettraient une première validation de chaque nouvelle version mais aussi un accroissement de la qualité du produit final en déchargeant l'équipe de validation des tests répétitifs.

2.3.2. Reproduction de bogue

La résolution de bogue est un problème récurrent dans le domaine informatique. Néanmoins, il n'est pas toujours des plus aisés de les traiter, en particulier dans les logiciels qui intègrent une grande partie de leur logique dans la communication inter machines. En effet, afin de trouver une solution à un problème de fonctionnement, il faut en toute logique d'abord en trouver la source afin d'agir sur celle-ci. Pour cela, les informaticiens enregistrent toutes les activités et états logiques du logiciel afin de cerner au plus près les conditions qui entraînent un dysfonctionnement du logiciel.

Il faut savoir également que certains bugs sont extrêmement difficiles à cerner car se déclarant dans des conditions précises d'utilisation difficiles à mettre en évidence, ou même parfois de manière qui semble aléatoire. Il devient alors ardu de le corriger car il est impossible pour l'informaticien de le reproduire à plusieurs reprises afin de bien saisir sa provenance et d'agir ensuite sur celle-ci.

De manière similaire les bogues des logiciels qui intègrent des communications inter machines sont difficiles à reproduire car ils obligent les équipes de développement à recréer des conditions de dysfonctionnement parfois complexes parce que extrêmement spécifiques. En effet, il peut suffire par exemple d'un délai de réponse trop long de quelques centièmes de seconde pour que le bogue se déclenche, ou bien d'un événement particulier déclenchant le problème de fonctionnement. On peut également imaginer qu'un bogue ne se déclenche qu'à la suite d'une séquence d'événements précis et difficiles à identifier. De plus, avec ce type de logiciels, il est plus difficile d'enregistrer toutes les informations nécessaires mais également de les recouper entre elles (du fait la répartition de fichiers sur plusieurs machines).

Enfin, malgré les nombreux tests qu'un logiciel subit avant d'être mis en production, il est utopique de croire que celui-ci ne contient plus de bogue. En effet, certaines situations particulières peuvent être rencontrées dans des cas extrêmes, que l'on n'a pas testés ou que l'on n'a tout simplement pas imaginés lors du développement. Ces bogues, qui peuvent être intrinsèquement liés à l'environnement de production, sont difficilement reproductibles et sont qui plus est mal documentés (en effet, lors de la mise en production, il n'est pas rare de désactiver tout ou partie des traces du système afin de gagner en performances et de ne pas générer des informations inutiles en cas de bon fonctionnement).

On devine aisément ici les cas d'utilisation qu'un système de rejeu et les avantages qu'ils procurent. En effet, il suffirait de capturer les activités réseaux à l'aide d'Ethereal lors d'un dysfonctionnement afin de pouvoir reproduire celui-ci autant de fois qu'il est nécessaire. De plus, cette trace pourrait dans une certaine mesure être intégrée dans les tests réalisés afin de s'assurer que l'on n'a pas réintroduit le bogue dans des versions ultérieures.

Un autre avantage est la possibilité d'enregistrer les bogues trouvés chez un client sur une machine de production, et de les reproduire sur une machine de développement. Cela permettrait aux développeurs de rechercher la source du problème en limitant les désagréments pour le client.

2.3.3. Optimisation des paramètres

La Proxy Platform comporte un nombre impressionnant de paramètres. Il est parfois difficile d'évaluer les conséquences de chacun sur les performances et/ou la stabilité du système. De plus, ces paramètres ne sont pas indépendants les uns des autres. Il devient alors difficile de trouver la configuration optimale que ce soit au niveau performance ou stabilité, suivant le cas de figure que l'on souhaite favoriser.

Le rejeu pourrait ici trouver une autre utilité en permettant de simuler plusieurs fois la même activité afin de trouver les paramètres adéquats aux cas d'utilisation que l'on favorise.

2.4 Spécifications du rejeu

Dans cette section, nous allons aborder les besoins spécifiques du rejeu. Nous allons commencer par spécifier la provenance des traces à rejouer. Ensuite nous allons indiquer clairement les critères auxquels la solution devra répondre.

2.4.1. Capture de traces

Avant de pouvoir rejouer les traces, il est évident qu'il faut préalablement les capturer.

Il était nécessaire dans un premier temps de définir les limites du rejeu. En d'autres termes, il nous fallait spécifier l'endroit des captures des traces. En effet, plusieurs choix s'offraient à nous comme le montre le schéma suivant.

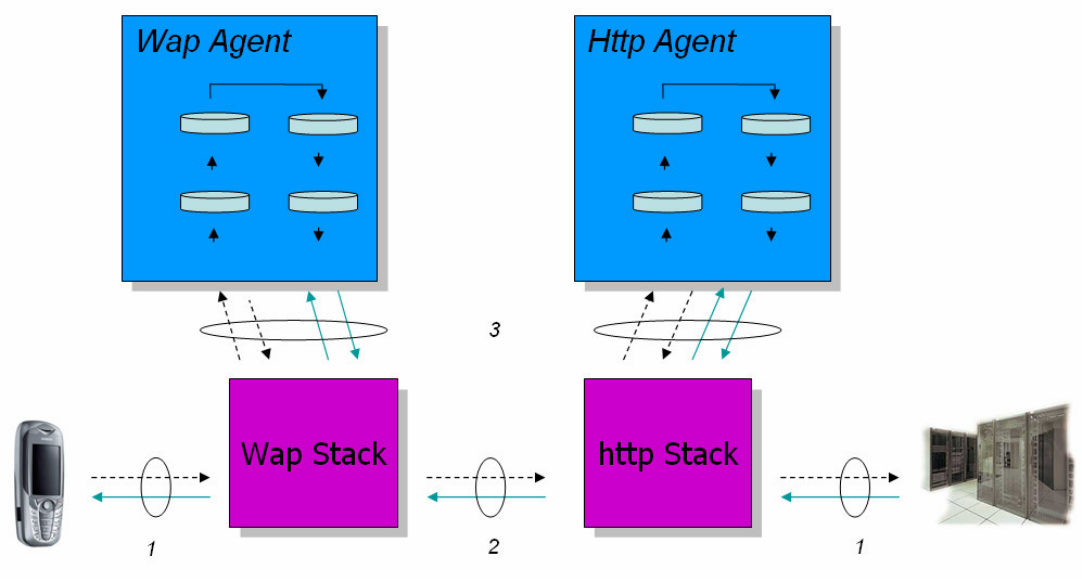

FIG 2.2 - Illustration des possibilités de capture de traces

Dès le départ, il nous semblait logique de capturer les traces de chaque côté du composant afin d'isoler celui-ci et d'avoir un contrôle complet. Cette première option implique une capture de traces à l'entrée et à la sortie de la Proxy Platform et donc d'une manière plus générale, la capture des trames échangées avec l'extérieur (captures 1 dans la figure 2.2). Ceci permet donc de simuler un cas d'utilisation de la Proxy Platform en l'isolant du monde extérieur et en récréant celui-ci. Cette option se base donc sur deux fichiers de traces.

La deuxième solution est plus riche (captures 1 et 2 dans la figure 2.2) car elle combine les traces de la première solution avec celles prises entre chaque stack. Cette solution a donc l'avantage de permettre un contrôle plus fort lors du rejeu, mais également la possibilité d'isoler la « simulation » à un seul couple stack – agent. Cette solution propose donc la capture d'un fichier de traces supplémentaires.

Quant à la troisième option, elle est encore plus précise (captures 1, 2 et 3 dans la figure 2.2). Elle permet d'isoler non plus un couple stack – agent mais un composant unique. On pourrait donc dans ce cas de figure, isoler par exemple la WAP stack afin de lui soumettre un scénario particulier. Remarquons que dans ce cas, il est nécessaire d'utiliser trois fichiers pour permettre le rejeu, au lieu de deux dans les autres options, car la WAP stack interagit avec 3 composants différents : le client, l'agent et enfin la stack HTTP.

Finalement, nous avons retenu la première option car il ne nous a pas semblé primordial de limiter la « simulation à un seul composant. En outre, la capture de traces à l'entrée et sortie de la Proxy Platform nous semblait l'approche la plus logique.

2.4.2. Description de la fonctionnalité de capture de traces WAP

Dans un souci de faciliter l'utilisation du rejeu mais également afin de standardiser le contenu et les fichiers utilisés, il a été nécessaire de développer dans l'interface web existante un nouveau dispositif permettant de capturer facilement les traces relatives à l'activité WAP. En effet, comme dit précédemment, la *Proxy Platform* est un système distribué ; il est donc impossible de savoir à l'avance quelle machine va traiter les requêtes du mobile que nous souhaitons enregistrer.

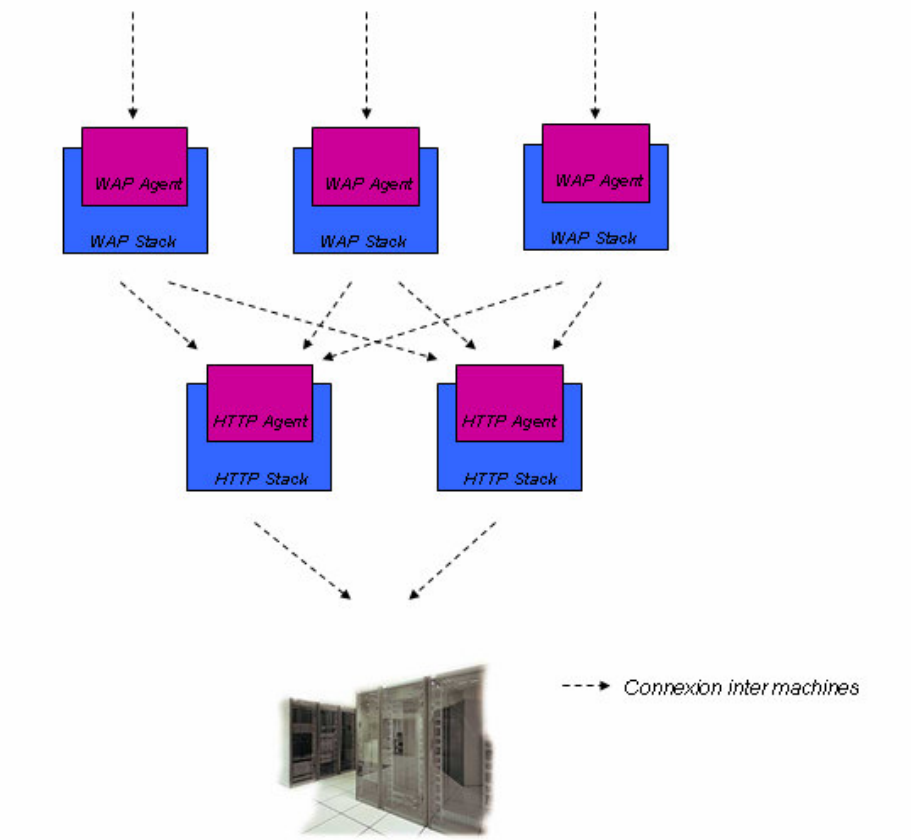

FIG 2.3 – Illustration d'une implémentation distribuée de la passerelle WAP

A partir de l'interface web, l'utilisateur est invité à introduire le numéro de mobile ainsi que le répertoire de sauvegarde des traces qui vont être capturées. Dès que l'utilisateur clique sur le bouton démarrer de l'interface, toutes les machines en cours d'utilisation vont recevoir l'ordre d'exécuter une nouvelle instance de la classe EtherealManager. Ces instances vont sonder une table particulière dans le système de la *Proxy Platform*. En effet, et comme expliqué au chapitre 1, un mobile doit nécessairement obtenir une adresse IP avant de pouvoir bénéficier des services WAP. Dès qu'une entrée dans la table correspond au numéro de mobile, les instances d'EtherealManager vont démarrer la capture de traces avec un filtrage optimal (capture des seules traces utiles au rejeu).

Néanmoins, ce système montre ses limites assez rapidement. En effet, il n'est pas rare de constater dans le fichier de traces que les premières trames de la transaction manquent. Cela est dû au temps nécessaire à Tethereal (version console d'Ethereal ayant les mêmes fonctionnalités) pour démarrer la capture effective (meilleur des cas : 0,1 seconde ; pire cas : 1,5 seconde). Afin de pouvoir capturer ces trames, il a été nécessaire de les enregistrer dans un fichier log en attendant le démarrage effectif de Tethereal. Malheureusement, ce fichier log

contient moins d'informations sur les trames échangées que le format *libpcap* utilisé par Tethereal. En effet, seules les données brutes sont enregistrées ainsi que les adresses IP des machines source et destination.

2.4.3. Spécifications

Nous allons maintenant aborder les principaux critères à respecter concernant le rejeu à partir de traces Ethereal.

Le but du rejeu de traces Ethereal est d'émuler le comportement d'un terminal ou d'un serveur interagissant avec la *Proxy Platform* afin de pouvoir tester de manière rigoureuse et facile le comportement de celle-ci. Il est évident que l'émulation d'un mobile utilisant des fonctionnalités WAP n'est qu'un cas d'utilisation et que le système de rejeu ne doit pas se limiter uniquement à celui-ci. Notre choix s'est porté sur ce cas d'utilisation car il utilise les deux protocoles les plus employés sur Internet, c'est-à-dire l'UDP qui sera utilisé dans la partie client de l'émulation ; et le TCP qui sera utilisé dans la partie serveur de l'émulation.

D'emblée, nous avons énoncé différents critères qui ont composé le cahier des charges auquel le système de rejeu à développer devait répondre :

- ✓ **Le contrôle du contenu des paquets**.

  Ce critère propose de vérifier le contenu de chaque paquet reçu avec celui que l'on attendait en sachant que les paquets émis doivent être rigoureusement identiques aux originaux. Il est donc nécessaire de mettre au point un mécanisme permettant de s'assurer de la correspondance entre un paquet reçu et celui que l'on attend. Ce mécanisme est évidemment différent suivant le protocole que l'on souhaite rejouer, mais il nous paraît également intéressant de pouvoir le modifier sans trop de difficultés (par exemple, modifier le niveau de rigueur du test). Ce critère est primordial.

- ✓ **Le respect du nombre de trames émises et reçues.**

  Ce critère consiste à tenter de respecter au mieux le formatage des données sur le réseau, plus que le contenu lui-même. Néanmoins c'est un critère difficile à respecter parce qu'intrinsèquement lié à l'environnement de travail. En effet, la taille maximum des paquets varie fortement suivant le type de connexion utilisée, mais également suivant l'OS utilisé ou encore le fait de travailler en loopback (MTU : 16634) ou via un réseau réel (MTU : 1500). Il nous paraît évident que ce critère ne sera jamais

complètement rempli, mais il a été convenu de s'en rapprocher le plus possible, le principal étant évidemment que les données transmises soient rigoureusement identiques aux originales (critère précédent).

- ✓ **Le respect du timing entre les paquets.**

  Ce critère temporel consiste à respecter au plus juste le temps écoulé entre chaque émission de paquet. Il nous est effectivement impossible d'agir sur les temps d'attente de réponse en tant que tels, bien que, comme nous l'avons précédemment dit, la solution adoptée envisage un contrôle de part et d'autre de la Proxy Platform, et donc un certain contrôle sur tous les temps d'attente. Ce critère, qui ne faisait pas partie de la première spécification, nous a paru néanmoins important et a donc été pris en compte assez tôt. De plus, il ne présente a priori aucune difficulté insurmontable.

Nous remarquons que tous ces critères ne s'excluent pas et qu'ils sont même facilement complémentaires. Néanmoins, suivant les solutions proposées, décrites au chapitre suivant, nous verrons que nous apporterons plus d'importance à certains d'entre eux.

3. Solutions proposées

3.1 Introduction

Ce chapitre va décrire les contraintes relatives au rejeu à partir de traces de la partie client et de la partie serveur. En effet, nous allons exposer dans ce chapitre les différents problèmes rencontrés et les solutions apportées afin d'atteindre les objectifs décrits au chapitre 2. Comme dit précédemment, certains de ces objectifs fixés ont été parfois non atteints dans la configuration actuelle. Néanmoins la suite de ce mémoire envisagera de nouvelles solutions afin de se rapprocher le plus possible du cahier des charges.

Les trois premières sections de ce chapitre vont nous permettre d'introduire la suite d'outils utilisés, tcpreplay mais également de faire un bref rappel de la composition d'une trame. Ensuite, les deux sections suivantes détailleront les contraintes liées aux problèmes de rejeu, auxquelles nous avons été confronté. Enfin la dernière section expliquera les limitations rencontrées dans les solutions développées.

3.1.1. Deux solutions distinctes

Nous avons vu qu'Ethereal ainsi que d'autres outils du même type (tcpdump, snort,…) sont capables de capturer toute l'activité réseau d'une machine et de la sauvegarder dans un fichier. Afin de pouvoir reproduire les trames contenues dans ce fichier il est évident qu'il faut préalablement lire le format *libpcap* dans lequel celles-ci sont stockées. Deux solutions sont possibles :

- ✓ Utiliser un outil capable de lire ces données et de les réinjecter directement sur le réseau. Il se trouve qu'un logiciel du nom de tcpreplay a été développé dans cette optique et qu'il est distribué sous une licence de type BSD. Il est à noter également que

cette fonctionnalité pourrait être intégrée dans Ethereal prochainement (Source : la mailing liste d'Ethereal)

- ✓ Trouver un outil capable de lire ce format et d'en extraire les données qui nous intéressent c'est – à – dire les données de la couche application. Cette fonctionnalité est également proposée par tcpreplay.

En fait, ces deux solutions ont été suivies car elles répondent à des besoins différents. En effet, le rejeu du côté client a adopté la première solution et le rejeu côté serveur la seconde.

La première solution envisage l'emploi d'un outil spécifique, en l'occurrence tcpreplay, pour la réémission des trames. Il est donc nécessaire de coordonner les activités de ce programme afin d'émettre les trames au moment voulu.

3.1.2. Tcpreplay

Tcpreplay est une suite d'outils sous licence BSD développée originellement par Dug Song et Matt Undy et maintenu à présent par Aaron Turner pour les systèmes UNIX. Elle permet d'utiliser des traces préalablement capturées au format *libpcap* dans le but de tester une grande variété de dispositifs. Ces outils sont en fait des applications : tcpprep, tcpreplay, flowreplay. [Turner]

Ces outils permettent de séparer le flux entre client et serveur via 2 interfaces réseaux différentes, de réécrire les couches réseaux 2, 3 et 4 partiellement ou totalement et finalement de réémettre des trames sur le réseau tout en recalculant les sommes de contrôle.

Tcpprep permet de parcourir le fichier de traces afin de préparer le rejeu. Pour cela, il génère un fichier que l'on passera en arguments à tcpreplay. Flowreplay quant à lui permet de rejouer un flux contenu dans un fichier de traces. Il est en cours de développement et n'est pas encore stable.

La suite d'outils tcpreplay permet le rejeu de traces quelconques à des vitesses arbitraires. Son objectif principal est le test de performance d'un NIDS (Network Intrusion Detection System) en rejouant un trafic d'arrière plan dans lequel il est possible de cacher des

attaques. Son avantage par rapport au trafic généré par logiciel est de reproduire exactement les conditions d'un trafic réel avec ses anomalies comme des routes asymétriques, des occupations de bande passante variables dans le temps, des fragmentations,…

3.1.2.1. Modifications apportées à tcpreplay

Quelques modifications ont été apportées à tcpreplay. La première concerne l'option –D qui permettait l'extraction des données contenues dans les trames du fichier de traces. Ces données étaient ensuite imprimées dans la console. Cette fonctionnalité a été modifiée afin de permettre l'écriture de ces données dans un fichier différent pour chaque trame ; ces fichiers étant regroupés dans un répertoire défini en argument.

La seconde modification concerne la fonctionnalité qui permettait de rejouer les trames une par une en mode console. En effet, si l'on démarrait l'application avec l'option -1, le programme demandait pour chaque trame la confirmation via la console. Le comportement de l'application a été modifié afin de répondre non plus à une frappe au clavier mais au message SIGUSR2[2]. Cela permettra au processus Java de contrôler l'envoi de trames de tcpreplay. Il est évident que le rejeu doit utiliser cette version 2.31 modifiée de tcpreplay.

3.1.3. Composition d'une trame

Pour rappel, une trame est toujours composée de plusieurs entêtes suivies des données transmises proprement dites. Dans la figure 3.1 et la figure 3.2, on constate donc qu'elle est composée de 3 entêtes : celui de l'Ethernet, de la couche réseau (IP) et enfin de la couche transport (TCP/UDP).

[2] Mécanisme permettant d'avertir un processus d'un événement extérieur dans un environnement linux

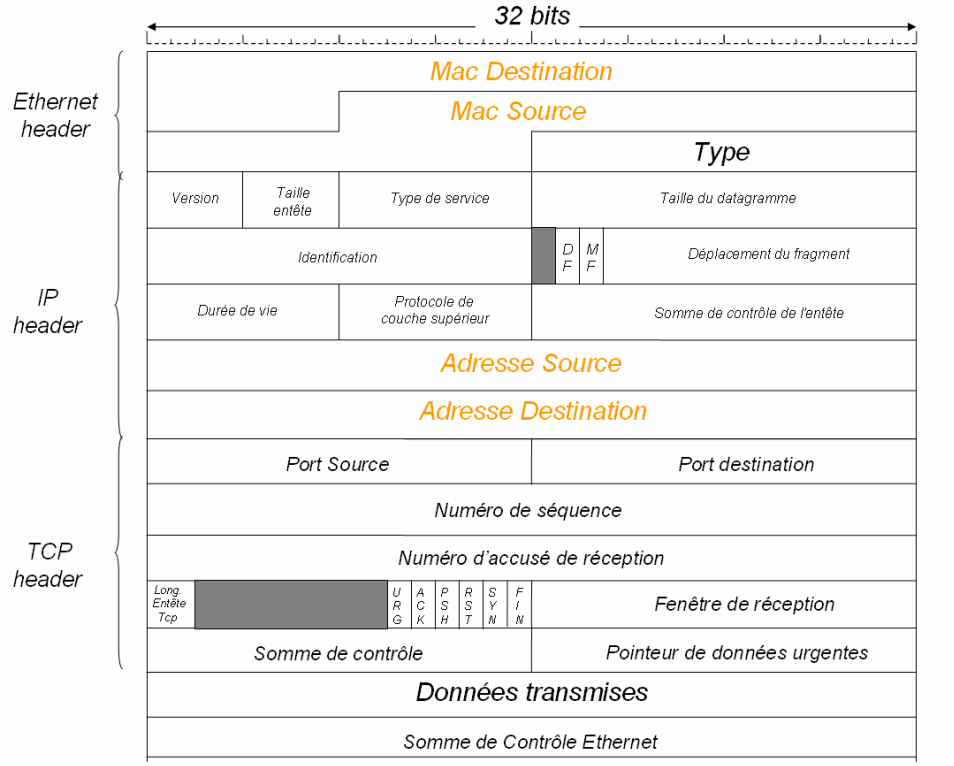

FIG 3.1 – illustration des différentes entêtes contenues dans une trame TCP

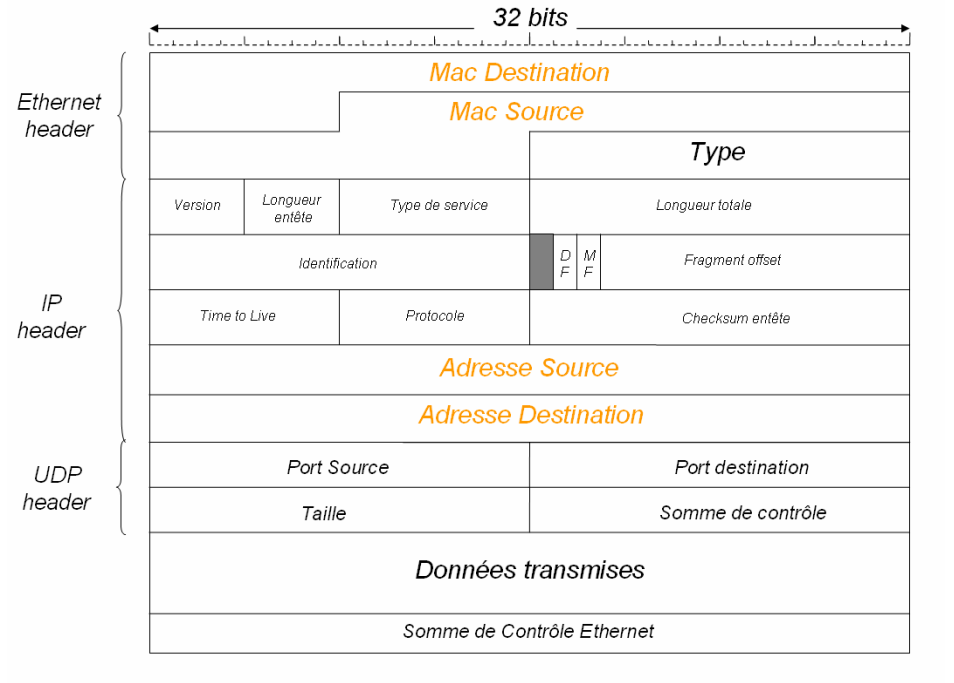

FIG 3.2 – illustration des différentes entêtes contenues dans une trame UDP

3.2 Emulation client

Cette section va introduire le lecteur aux contraintes liées à l'émulation client. Bien que ces contraintes ne soient pas réellement spécifiques à l'émulation client puisqu'elles se retrouvent également dans l'émulation serveur, nous allons les spécifier dans cette section ainsi que leurs solutions respectives. Nous commencerons par les contraintes d'adressage, les contraintes d'authentification de trames pour finir avec les contraintes de synchronisation requête / réponse. La dernière section détaillera la solution développée au cours du stage.

3.2.1. Introduction

Nous allons détailler dans cette section la solution apportée pour le rejeu d'une transaction WAP entre un mobile et la *Proxy Platform* sur base de traces capturées à l'aide d'Ethereal. Nous allons substituer le mobile par une application émulant son comportement comme on peut le voir sur la figure 3.3.

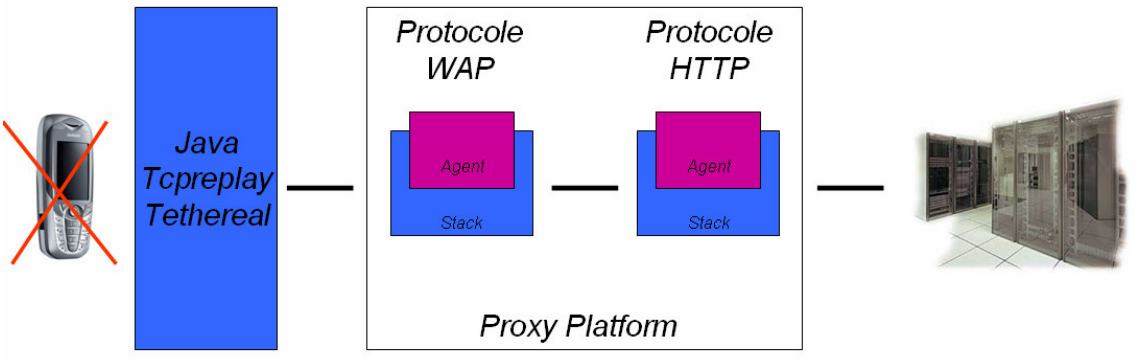

FIG 3.3 – Illustration de l'émulation client

3.2.2. Contraintes d'adressage

Comme nous le voyons sur les figures 3.1 et 3.2, l'entête IP comprend 2 champs de 4 octets contenant l'adresse IP source et l'adresse IP destination. De manière analogue, l'entête Ethernet contient 2 champs de 6 octets contenant les adresses MAC de la machine destinataire et de la machine source.

Lors du rejeu, ces informations ne correspondent plus à la réalité. En effet, comme nous l'avons dit plutôt, les traces à rejouer proviennent peut – être d'une société cliente et donc ces informations n'ont pas de sens dans le contexte dans lequel on va les rejouer. De plus, il nous paraît judicieux d'avoir un contrôle plus important sur ces données afin de pouvoir rejouer les traces dans différents environnements (Nous entendons par là des machines avec des adresses MAC et IP différentes).

Il devient donc nécessaire de réécrire ces données à chaque exécution du rejeu afin de prendre en compte l'environnement dans lequel la machine se trouve. Dans cette optique, il sera obligatoire de fournir au programme toutes ces informations avant de pouvoir commencer le rejeu.

Une nouvelle fois, cette réécriture fait partie des fonctionnalités de tcpreplay. Néanmoins, ceci limite le rejeu à l'émulation d'un seul terminal. En effet, la fonctionnalité développée dans tcpreplay permet la réécriture des couples d'adresses MAC contenus dans le fichier de traces par un seul couple d'adresses. Cette solution s'est avérée cependant largement suffisante car notre objectif se limite à rejouer un seul terminal à la fois.

3.2.3. Contraintes d'analyse de trames

Nous allons aborder maintenant le problème de détection des trames relatives à un client et celles relatives à un serveur. En effet, lors de la capture de traces avec Ethereal, bien que l'on ait utilisé les filtres adéquats afin de ne récupérer que les traces qui nous intéressaient, il faut trouver un moyen de séparer ces traces en 2 flux distincts afin de savoir quelles trames nous allons envoyer en tant que client et par déduction quelles trames nous allons attendre.

Ici apparaît une première différence entre le traitement des protocoles basés sur UDP et ceux basés sur TCP. En effet, vu la politique orientée sans connexion de l'UDP, il est extrêmement difficile de discerner une trame provenant d'un client de celle provenant d'un serveur, sur base des seules traces. Au contraire, en ce qui concerne TCP, il existe une approche plus efficace.

3.2.3.1. UDP

Comme nous pouvons le voir sur la figure 3.2, l'entête UDP est très légère, nous ne disposons donc que des informations telles que les adresses IP et les ports utilisés pour faire notre tri entre trame client et trame serveur.

Remarquons que certains protocoles basés sur UDP utilisent des ports spécifiques pour leurs activités. C'est le cas notamment du protocole WAP, qui utilise spécifiquement 4 ports : 9200, 9201, 9202, 9203. Dans la spécification du WAP, le port 9200 est utilisé pour un mode de fonctionnement orienté sans connexion tandis que le 9201 est utilisé pour un mode de fonctionnement orienté connexion. Les ports 9202 et 9203 sont quant à eux utilisés pour des connexions WTLS sécurisées avec à nouveau une distinction orientée connexion et sans. Donc, a priori et en ce qui concerne le WAP, il est facile de détecter les trames relatives à un serveur de celles relatives à un client en filtrant les trames sur ce critère de port utilisé.

Néanmoins pour d'autres protocoles comme notamment le SIP, il n'existe pas de port spécifique sur lequel on peut se baser. Dans ce cas de figure, il sera nécessaire de demander à l'utilisateur l'adresse IP du client dans la trace afin de pouvoir trier ses propres trames.

3.2.3.2. TCP

En ce qui concerne le TCP, le tri des trames est plus facile car il existe plusieurs approches possibles. Remarquons que les techniques utilisées pour l'UDP sont également applicables pour le TCP, notamment le filtrage par adresse IP ou l'utilisation de port réservés (0 – 1024). En effet, par exemple, un serveur HTTP utilise souvent le port 80 ; toutes les trames utilisant ce port 80 comme port destinataire pourraient être considérées comme étant une trame émise par le client. D'ailleurs ce critère est utilisé par Ethereal pour sélectionner le dissecteur dédié au protocole HTTP.

Néanmoins, une autre approche est plus efficace. Comme nous le savons, le TCP est un protocole orienté connexion, et il utilise la méthode de connexion « *Hand – shaking* ». Cette méthode basée sur l'échange de 3 trames nous permet de facilement détecter un client ou un serveur.

Voici une représentation de cet échange :

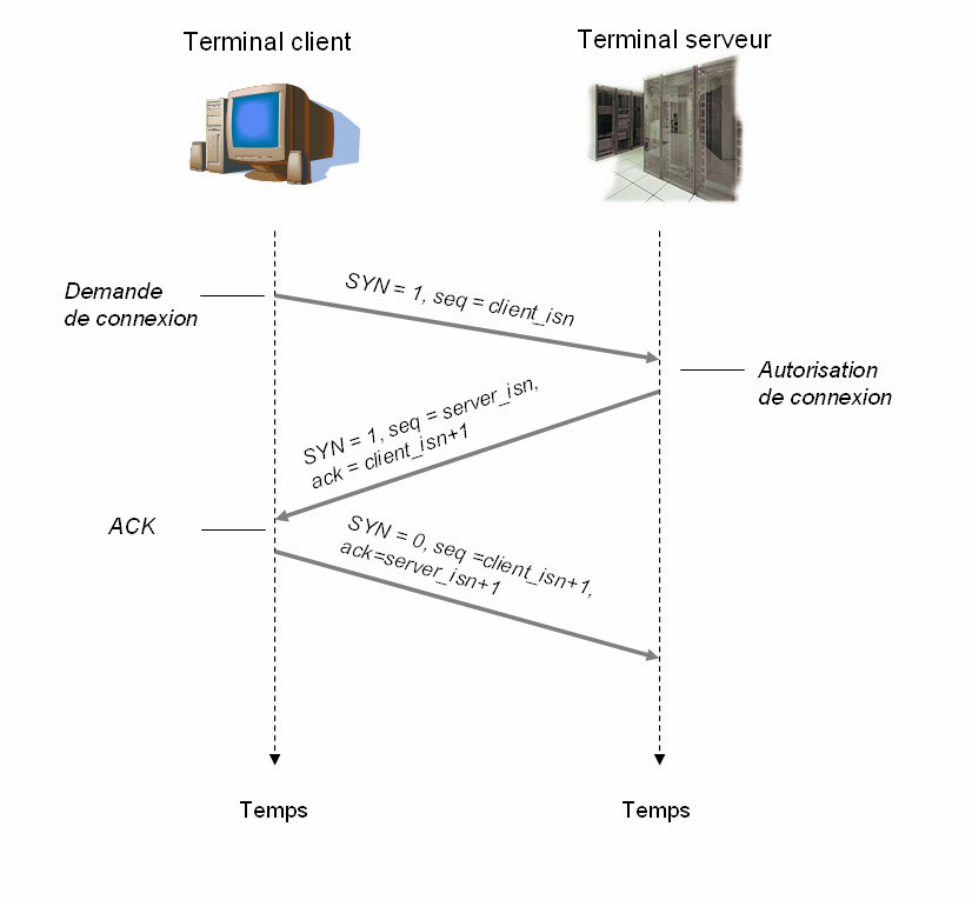

FIG 3.4 – Illustration du hand – shaking [Kurose&Ross]

Nous constatons sur la figure 3.4 que les deux premières trames échangées ont leurs fanions SYN positionnés à 1 (voir également la figure 3.1). Ce fanion contenu dans l'entête TCP nous permet d'identifier facilement un client. En effet, il suffit de vérifier le positionnement de ce fanion : s'il est positionné à 1 et que dans le même temps le fanion ACK est positionné à 0 alors nous sommes en train d'analyser la première trame de connexion. Nous pouvons donc récupérer l'adresse IP source et le port source utilisé dans cette trame en sachant que ces informations correspondent à un terminal client. De manière analogue, nous pouvons récupérer l'adresse IP destinataire et le port destinataire en sachant que ces informations sont celles d'un serveur.

Une fois identifiées les trames émises par le client dans le fichier de traces, nous pouvons commencer le rejeu.

3.2.3.3. Mécanisme d'authentification de trames

Comme nous l'avons décrit dans le chapitre 2, les trames reçues lors du rejeu doivent être comparées aux trames contenues dans le fichier de traces afin de s'assurer du bon déroulement du rejeu.

En effet, il est inutile de continuer le rejeu si celui-ci s'écarte trop du scénario original. Si lors du rejeu, un événement quelconque provoque un changement de scénario, comme par exemple l'émission d'une trame « abort » à la place de la réponse attendue, il est superflu de continuer le rejeu.

Il est donc nécessaire dans un premier temps de mettre en place un mécanisme de comparaison entre trame reçue et trame attendue.

Evidemment, il est vain de comparer les trames entre elles octet par octet. De fait, les informations contenues dans les champs des entêtes ont été partiellement réécrites et ne correspondent plus. De plus cette comparaison sévère souffre d'un manque de souplesse. Il a donc fallu trouver une autre méthode.

Avant tout chose, tâchons de définir ce que l'on considère comme étant deux trames équivalentes. Suivant les protocoles rejoués, il nous paraît opportun de pouvoir affiner cette équivalence suivant des critères différents. De plus, suivant le rejeu, il peut être intéressant de pouvoir changer la rigueur de ce test (l'assouplir ou le durcir).

Deux trames équivalentes sont deux trames dont le contenu sémantique est le même. Par exemple, en ce qui concerne la première trame (WSP Connect) envoyée lors de l'établissement d'une transaction WAP, on considère comme équivalente une trame de même type, donc identifiée par Ethereal comme étant une trame WSP Connect, contenant par exemple, selon le niveau de rigueur souhaité, les mêmes caractéristiques relatives aux mobiles.

Dans ce but, nous avons opté pour un système de parseur dédié. D'une part, il nous semblait que ce système permettrait à la fois la prise en charge facile de nouveaux protocoles et qu'il permettrait d'autre part de changer facilement la rigueur des tests d'équivalence. Le schéma suivant représente l'architecture du système employé. Il est suivi par des explications permettant d'en comprendre le fonctionnement.

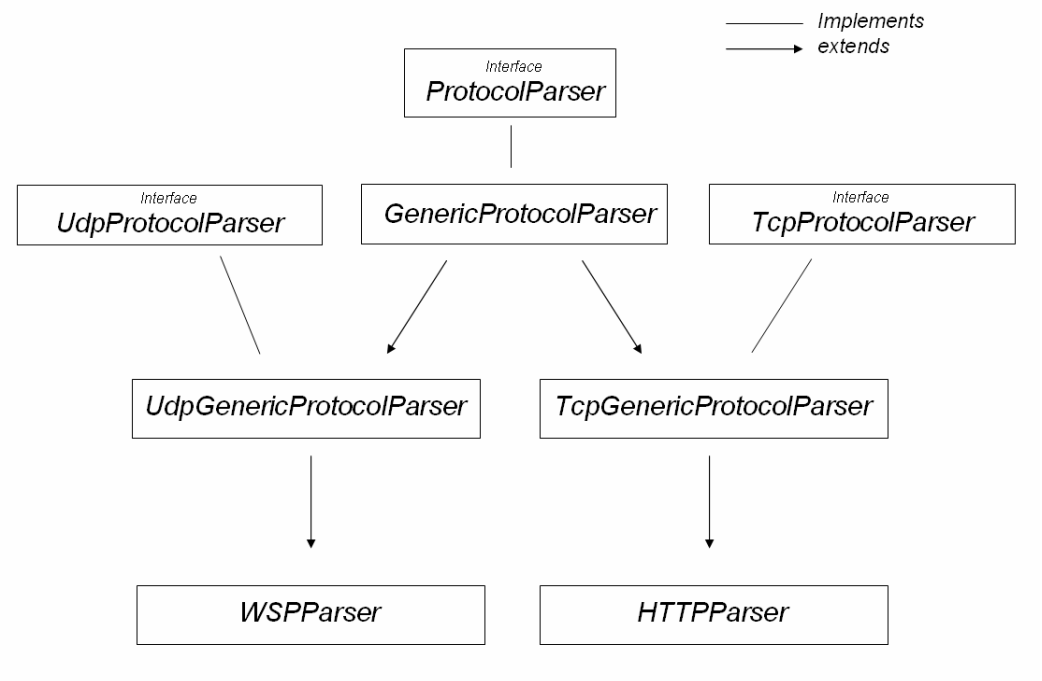

FIG 3.5 – Illustration de l'architecture du système de parseurs

La classe GenericProtocolParser est utilisée pour définir le comportement général des parseurs ; elle contient également toutes les méthodes d'extraction d'informations souvent utilisées comme les adresses IP, les ports utilisés mais aussi le protocole utilisé au niveau applicatif de la trame. Une fois cette dernière information trouvée, elle crée une nouvelle instance de la classe définissant le comportement à adopter pour ce protocole. Celle-ci va alors effectuer le travail spécifique propre au protocole c'est – à – dire extraire les données pertinentes afin de vérifier la concordance entre la trame reçue et la trame attendue.

Si la classe GenericProtocolParser ne trouve pas de classe dédiée au protocole identifié, la concordance entre la trame reçue et la trame attendue se fera sur base des informations déjà extraites (par défaut, c'est le contenu que l'on peut voir dans la colonne info de la première fenêtre d'Ethereal).

Lorsqu'on désire ajouter un nouveau protocole, il suffit d'étendre soit la classe UdpGenericProtocolParser, soit la classe TcpGenericProtocolParser suivant qu'il est basé sur UDP ou TCP. Cette classe permet de différencier le comportement général de l'application afin de l'adapter au besoin de ce nouveau protocole. Dès que l'on a implémenté les méthodes dérivées de l'interface voulue, il suffit de déclarer cette nouvelle classe ainsi que le nom du protocole qu'elle traite (le même nom utilisé par Ethereal) et cette classe sera utilisée à chaque fois que l'on rencontrera ce protocole dans un fichier de traces.

3.2.4. Contraintes de synchronisation requête / réponse

Maintenant que l'on a trié les trames à envoyer et les trames à recevoir et que l'on dispose d'un mécanisme de comparaison de trames, il ne reste plus qu'à suivre le scénario établi dans le fichier de traces. En effet, il est idiot d'envoyer la réponse à une requête avant l'avoir reçue.

Pour chaque trame, il suffit donc de déterminer si celle-ci doit être envoyée ou reçue avec la méthode expliquée dans ce chapitre. Si celle-ci doit être envoyée, on l'envoie à l'aide de tcpreplay en n'oubliant pas de réécrire les informations nécessaires, ensuite on traite la trame suivante dans le fichier de traces. Si celle-ci doit être reçue, on parse toutes les trames reçues et ce tant que l'on n'a pas trouvé une trame équivalente à celle attendue. Ensuite, on passe à la trame suivante dans le fichier de traces.

3.2.5. Description de la solution proposée

Nous allons décrire dans cette section la solution qui a été implémentée lors du stage dans la société Nextenso. Ensuite, dans la section 3.4.1, nous allons décrire les limitations rencontrées afin de pouvoir suggérer de nouvelles solutions dans le chapitre suivant.

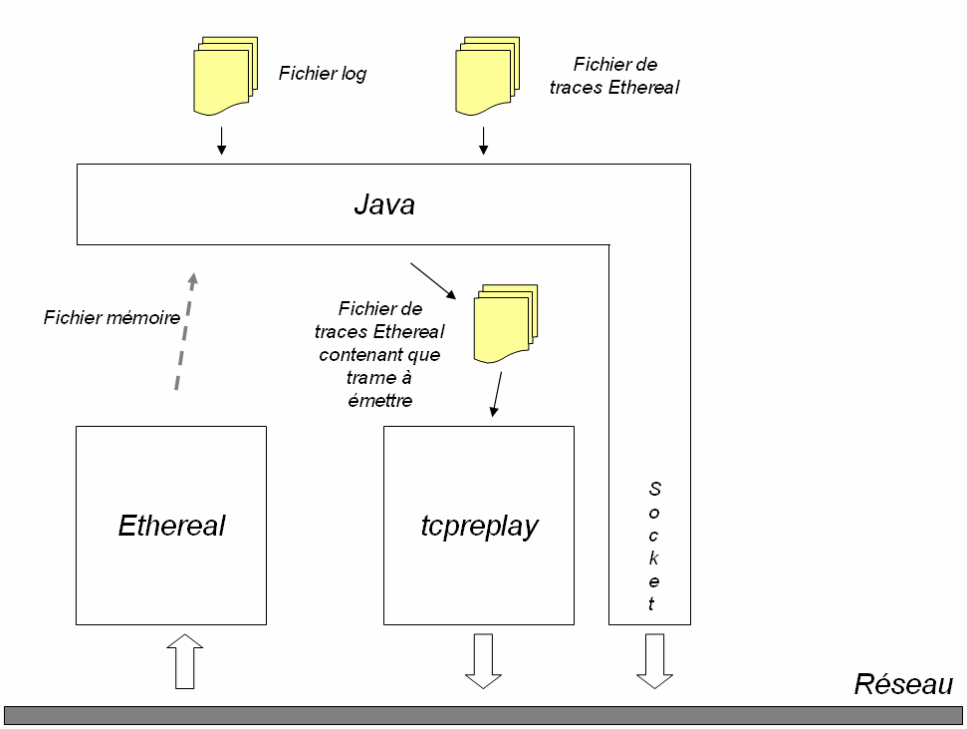

FIG 3.6 – Schéma de la solution complète proposée

Lorsqu'on démarre l'application, celle-ci effectue les traitements décrits plus tôt dans ce chapitre c'est – à – dire le filtrage des trames contenues dans le fichier de traces que l'on a reçu en argument. Ensuite le processus Java va démarrer une instance de Ethereal avec les arguments nécessaires afin que celui-ci traite les trames concernant le protocole que l'on veut rejouer. Le processus Java va également démarrer une instance de tcpreplay en lui donnant comme arguments un fichier au format *libpcap* contenant toutes les trames à envoyer durant le rejeu ainsi que toutes les informations précédemment décrites (les nouvelles adresses MAC, les nouvelles adresses IP) afin que celle-ci puisse les réécrire correctement.

Comme on peut le constater le rôle du processus Java est de coordonner les activités de tcpreplay et de tethereal, il doit aussi s'assurer des contraintes de correspondance entre trames reçues et trames attendues. C'est également lui qui va permettre de respecter les temps d'attente mesurés dans le fichier de traces.

En effet, pour chaque trame du fichier de traces, le processus Java va procéder à l'analyse de son contenu et déterminer si cette trame doit être émise ou reçue. Si elle est doit être envoyée, le processus Java va envoyer un signal à tcpReplay pour que celui – ci l'envoie. Au contraire, si la trame doit être reçue, alors le processus Java va analyser toutes les trames que

Tethereal lui a communiquées. Ce n'est qu'une fois ce test de correspondance satisfait que le processus Java passera à la trame suivante dans le fichier de traces.

On constate donc que ce système permet de suivre de trame en trame le scénario contenu dans le fichier de traces.

3.2.5.1. Démarrage alternatif

Comme nous l'avons vu au chapitre 2, il a été nécessaire d'enregistrer les premières trames relatives à une transaction WAP dans un fichier log. Nous devons donc commencer le rejeu en lisant ce fichier pour en extraire les trames manquantes dans le fichier de traces d'Ethereal. Vu que ce fichier contient moins d'indications sur le formatage des données sur le réseau, il est évident que celui-ci sera moins précis que le système décrit dans la section précédente. De plus, l'analyse des trames se faisant uniquement sur base des indices de ports utilisés et des adresses IP. Dans ce contexte, le basculement d'un système à l'autre doit se faire le plus tôt possible.

Etant donné que nous ne disposons que de données brutes, il est impossible d'utiliser tcpreplay. Il va donc falloir utiliser un autre moyen afin de permettre l'envoi de ces données sur le réseau. Nous allons utiliser des sockets Java de la librairie Java.Net. Les données lues en hexa décimal dans le fichier log sont reconverties en binaire, intégrées dans un datagramme UDP et envoyées par une socket standard dont les caractéristiques sont calquées sur celles contenues dans le fichier de traces. En effet, il est nécessaire d'utiliser le même port source dans tout le déroulement du rejeu.

Pour pouvoir basculer d'une méthode de rejeu à l'autre, il faut trouver un moyen de comparer les données contenues dans le fichier log et les données contenues dans le fichier de traces ; les deux fichiers n'ayant pas le même format. De plus, vu que le fichier log contient moins d'informations, nous devons nous baser uniquement sur les données applicatives, les adresses IP source et destination, les ports source et destinataire. Les adresses IP et les ports n'étant pas suffisants, nous devons donc comparer les données applicatives. Nous avons alors recours une fois de plus à tcpreplay afin d'extraire les données de la première trame contenue dans le fichier de traces Ethereal. Il ne reste plus qu'à comparer ces données avec celles trouvées dans le fichier log et de basculer si on trouve une correspondance.

3.3 Emulation serveur

Cette section va introduire le lecteur aux contraintes spécifiques à l'émulation serveur. Comme dit précédemment, les contraintes développées dans les premières sections de ce chapitre sont également applicables à l'émulation serveur. Néanmoins des contraintes supplémentaires doivent être prises en compte. Nous allons donc introduire dans cette section les contraintes spécifiques à l'émulation serveur, les contraintes de gestion de connexions parallèles et enfin les contraintes de gestion des sessions. La dernière section détaillera la solution développée au cours du stage.

3.3.1. Introduction

Nous allons détailler dans cette section la solution adoptée pour le rejeu d'une transaction HTTP entre la *Proxy Platform* et un serveur web quelconque sur base de traces capturées à l'aide d'Ethereal. Nous allons substituer le serveur web par une application émulant son comportement comme on peut le voir sur la figure 3.7.

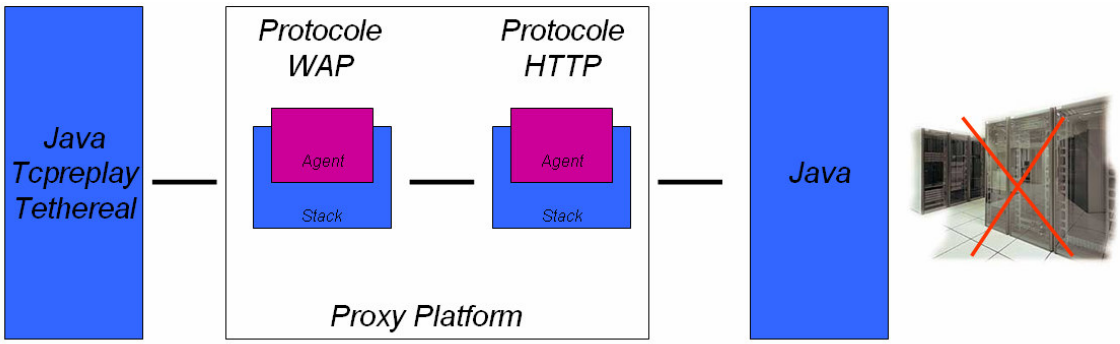

FIG 3.7 – Illustration de l'émulation serveur

3.3.2. Contraintes propres aux serveurs

Un serveur est par définition un ordinateur détenant des ressources particulières qu'il met à la disposition d'autres ordinateurs par l'intermédiaire d'un réseau. Il n'est donc pas l'initiateur de la demande de service, il doit répondre à des demandes. Or, nous voyons dans la

figure 3.7, que les requêtes proviennent de la *Proxy Platform* et plus particulièrement de la stack HTTP. Nous n'avons donc aucun contrôle sur les différents paramètres (numéro d'acquittement, numéro de séquence,…) de la requête et nous devons les accepter comme ils nous parviennent et les intégrer dans la réponse afin que la stack accepte celle-ci. En effet, si les numéros de séquence et/ou les numéros d'acquittement sont erronés, la stack HTTP ne va pas accepter ces réponses et renouveler sa requête.

Deux solutions sont alors envisageables. La première envisage l'adaptation de ces trames au contexte du rejeu (nouvelles adresses MAC et IP, éventuellement nouveaux indices de port) avant de les mettre sur le réseau. La deuxième, quant à elle, envisage l'extraction des données de la couche applicative des trames contenues dans le fichier de traces et l'utilisation de sockets de la librairie Java.Net permettant une gestion automatique de la signalisation de la connexion TCP (numéro de séquence, taille fenêtre,…). Nous avons opté pour la seconde solution car la première solution nécessitait l'ajout d'une fonctionnalité dans tcpreplay permettant de réécrire de nouvelles informations (numéro de séquence, numéro d'acquittement). De plus, ces informations étant différentes pour chaque trame, il fallait également implémenter un mécanisme permettant de communiquer ces informations pour chaque trame. Enfin, la seconde solution ne souffrait d'aucune contrainte de ce type et offrait même des avantages divers (gestion automatique de ces numéros, gestion aisée de plusieurs flux,…)

3.3.3. Contraintes de connexions parallèles

Une des particularités du protocole TCP sur lequel est basé le protocole HTTP, est qu'il permet de multiples connexions parallèles. En effet, le protocole TCP permet à un client de se connecter plusieurs fois simultanément à un même serveur et ce dans un but évident d'optimiser les temps de réponses. Cette particularité est souvent utilisée par le protocole HTTP qui, par exemple lors de l'affichage d'une page HTML, commence le chargement des images composant celle – ci avant que la page soit entièrement téléchargée. Il faudra donc que le rejeu puisse simuler ces accès concurrents.

Ces connexions parallèles et les requêtes qu'elles véhiculent peuvent arriver dans un ordre différent lors du rejeu que dans le scénario original. Les raisons de changement d'ordre sont multiples. Elles peuvent provenir du comportement de la *Proxy Platform* et de sa gestion des connexions parallèles, et par extension de la manière dont la priorité des threads est gérée. Elles peuvent provenir également du réseau lui – même. Par exemple, il suffit d'une erreur de

transmission pour retarder une de ces connexions parallèles et ainsi changer l'ordre d'arrivée. Néanmoins, il est important que ce changement d'ordre n'affecte pas le rejeu.

Toutefois, une autre contrainte apparaît. En effet, comme nous l'avons déjà remarqué, nous n'avons aucun contrôle sur les requêtes nous parvenant. Il est donc fort probable que la stack HTTP utilise des ports différents lors du rejeu que ceux contenus dans le fichier de traces. Ce qui implique l'implémentation d'un système permettant de faire correspondre les ports originaux avec ceux du rejeu. En effet, vu que l'ordre de réception des requêtes peut être différent, il est inutile de suivre à l'aveuglette le scénario contenu dans le fichier de traces. Il faut donc implémenter un système permettant de recevoir les requêtes dans un ordre différent, d'identifier les réponses correspondant à ces requêtes dans le fichier de trace et de les renvoyer sur le port adéquat. De fait, il est indispensable de renvoyer la réponse correspondant à la requête reçue.

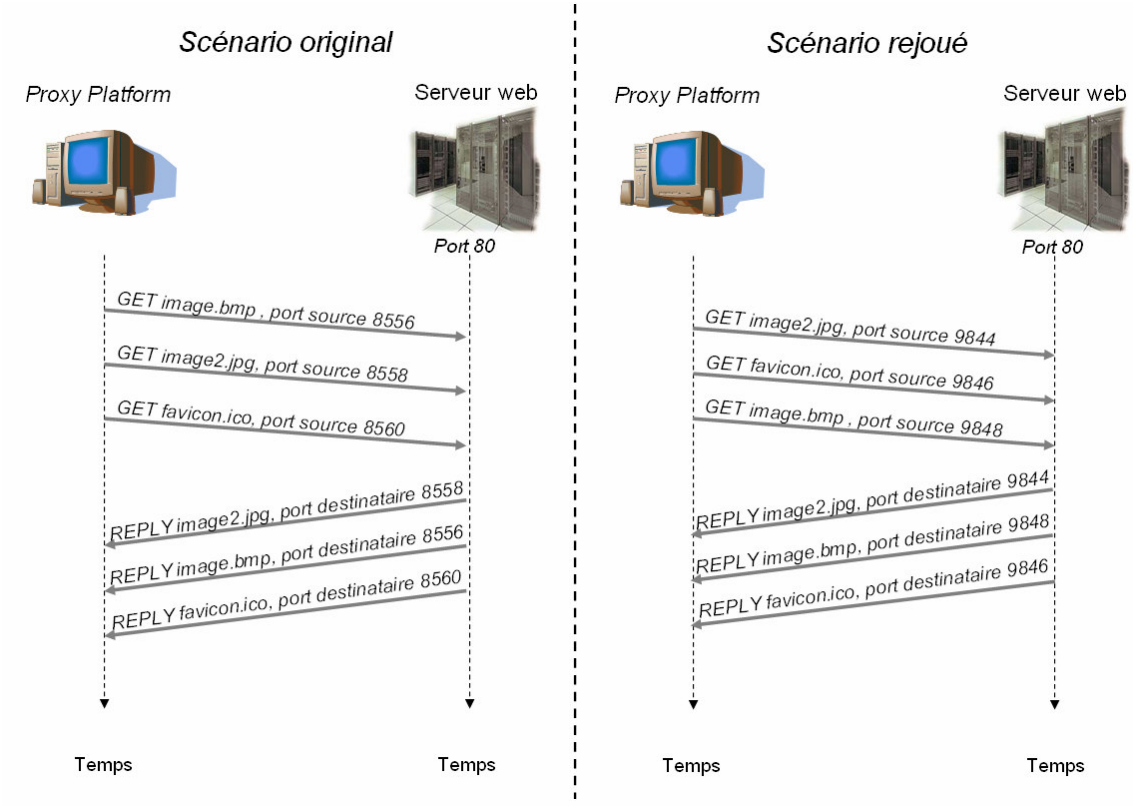

FIG 3.8 – Illustration de la contrainte de connexions parallèles.

L'implémentation que nous avons développée permet de recevoir les requêtes dans un ordre quelconque mais les réponses seront envoyées dans le même ordre que dans le scénario original comme le montre la figure 3.8.

Pour commencer, il faut savoir que chaque requête reçue est mémorisée sous la forme d'un couple (intitulé de la requête - port utilisé par le client). Dans notre exemple, trois couples vont être créés à la réception des trois requêtes :

- (GET image2.jpg – 9844)
- (GET favicon.ico – 9846)
- (GET image.bmp – 9848)

Il y a donc une socket qui écoute en permanence sur le port 80 à l'instar d'un serveur web réel. Lorsqu'on lit dans le fichier de traces une trame contenant une requête, on la compare à celle déjà reçue. Soit celle-ci s'y trouve déjà, on crée alors un nouveau couple (port utilisé dans le fichier de trace – port utilisé dans le rejeu) et on passe à l'analyse de la trame suivante dans le fichier de traces, Soit celle-ci n'a pas encore été reçue et on va attendre son arrivée avant de continuer le traitement. Dans notre exemple, trois nouveaux couples vont être créés :

- (GET image.bmp – 8556) + (GET image.bmp – 9848)=(8556 – 9848)
- (GET image2.jpg – 8558) + (GET image2.jpg – 9844)=(8558 – 9844)
- (GET favicon.ico – 8560) + (GET favicon.ico – 9846)=(8560 – 9846)

On voit dans notre exemple que l'on va attendre les trois requêtes avant de commencer à envoyer les réponses. Lors de l'envoi des réponses, il suffira de consulter les couples (port utilisé dans le fichier de traces – port utilisé dans le rejeu) afin d'envoyer la réponse sur le bon port client.

Comme on peut le constater, la particularité de ce système est de renvoyer les réponses dans le même ordre que le scénario original bien que les requêtes arrivent potentiellement dans un ordre différent.

3.3.4. Contraintes de gestion des sessions

Le protocole TCP est un protocole intégrant la gestion de sessions. En effet, il gère l'ouverture et la fermeture de session. L'ouverture de session s'effectue par l'échange de 3 trames et s'appelle le « *Hand – shaking* ». Vu que cet échange a déjà été décrit dans les premières sections de ce chapitre, nous n'allons plus le détailler ici. En contrepartie, la procédure de fermeture n'a pas encore été décrite, nous allons la décrire brièvement.

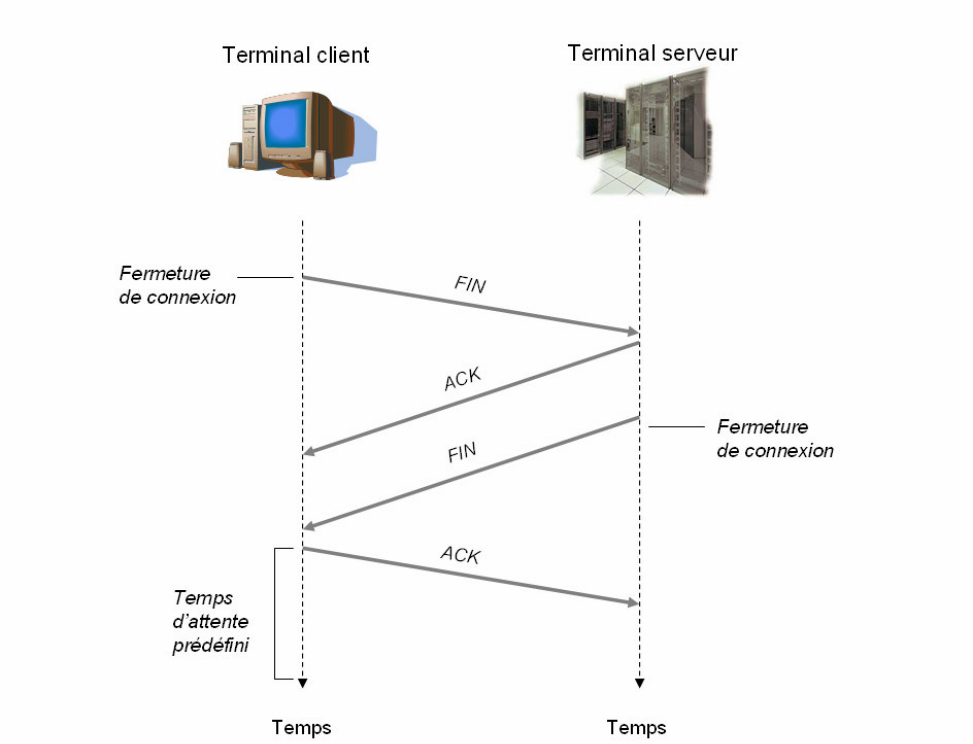

FIG 3.9 – illustration de la fermeture de session TCP.

La procédure est décomposée en 4 étapes : le client qui décide de fermer la session envoie une trame avec son fanion FIN positionné sur 1 (voir figure 3.1), le serveur acquitte cette trame et envoie également une trame avec son fanion FIN activé. Une fois que les deux parties ont acquitté la trame, la session est fermée de part et d'autre. Il est à noter que le serveur peut également initialiser ce processus de fermeture de session. Il est donc nécessaire que le système de rejeu puisse également fermer les sessions.

D'autre part, le protocole TCP inclut également un mécanisme permettant de réutiliser une connexion ouverte (le *Keep – alive)* et ce dans le but de ne pas renouveler la procédure de connexion pour chaque requête. Il est évident que le système de rejeu devra également prendre en compte cet aspect.

3.3.5. Contraintes de flux multiples

Il est possible que dans le fichier de trace Ethereal, le client se soit connecté à plusieurs serveurs. En effet, un scénario simple d'utilisation WAP est de vérifier ses mails auprès de différents serveurs de mail. Il est donc primordial que le système de rejeu puisse émuler plusieurs serveurs. Ces serveurs peuvent utiliser des ports différents, il est donc nécessaire de détecter tous les ports serveurs utilisés dans le fichier de traces avant de commencer le rejeu.

Précisons également la manière dont l'émulateur serveur est intégrée dans le flux de communication. Afin de permettre l'interconnexion de plusieurs stacks, celles-ci prévoient un mode de connexion paramétrable. Lorsque l'on veut la stack HTTP interagisse directement avec le serveur impliqué dans la requête, on utilise le mot réservé DIRECT dans le fichier d'initialisation de la stack. Si l'on désire faire intervenir une autre stack dans le flux, ou bien dans le cas qui nous intéresse ici, l'émulateur serveur, il suffit d'initialiser la stack HTTP avec le couple « NOM_MACHINE PORT_UTILISE ». De cette manière, l'émulateur serveur sera intégré au flux de communication d'une manière analogue à celle d'un proxy.

3.3.6. Description de la solution proposée

Nous allons dans cette section décrire la solution qui a été implémenté lors du stage au sein de la société Nextenso. Ensuite, dans la section 3.4.2 nous allons décrire les limitations rencontrées afin de pouvoir dans le chapitre suivant suggérer de nouvelles solutions.

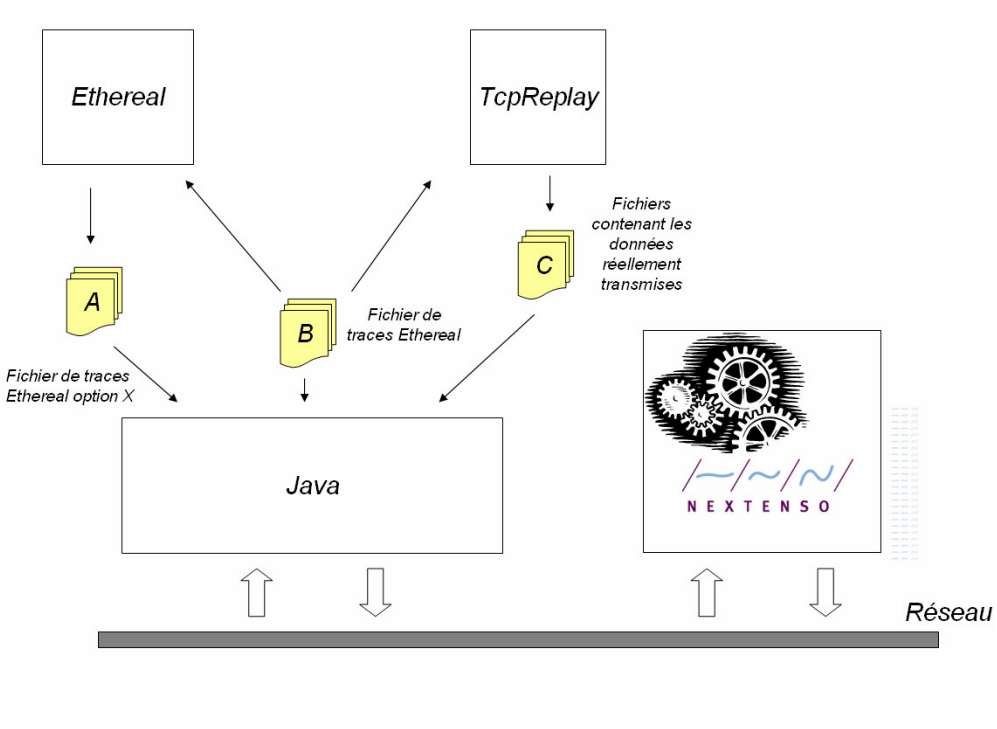

FIG 3.10 - Schéma de la solution serveur développé

Comme on peut le constater dans la figure 3.10, la solution est assez différente de celle développée pour l'émulation client (voir figure 3.6). Cela provient essentiellement du choix d'utiliser les sockets de la librairie Java.Net.

Le rôle de tcpreplay dans cette implémentation est différent de celui de l'émulation client. En effet, dans l'émulation client, on utilisait tcpreplay pour réémettre les trames sur le réseau. Ici son rôle est d'extraire les données contenues dans les trames du fichier de traces. Si l'on se réfère une nouvelle fois à la figure 3.1, cela revient à sauvegarder dans un fichier les données d'une trame à l'exclusion de toutes les entêtes. Dans cette optique, la fonctionnalité proposée par tcpreplay a été légèrement modifiée afin de permettre l'écriture du contenu de chaque trame dans un fichier différent ; ces fichiers étant regroupés dans un répertoire spécifié en argument.

Lors du démarrage de l'émulation serveur, le processus Java lance automatiquement un processus tcpreplay afin de préparer le rejeu. Ce processus va donc sauvegarder dans les fichiers (C) de la figure 3.10 les données transmises de chacune des trames comprises dans le fichier de traces (B) de la même figure. Dans le même temps, un processus Tethereal va parcourir le fichier de traces (B) afin de générer un autre fichier (A) contenant le détail de chaque trame. Ce

fichier détaillé (A) permettra l'analyse des trames et, avec un mécanisme similaire à l'émulation client, la distinction entre trame à envoyer et trame à recevoir.

Ensuite le processus Java instancie un Serversocket ; son rôle sera d'accepter toutes les connexions provenant du client et d'affecter le traitement à une classe spécialisée. En effet, pour chaque protocole utilisé en tant que serveur, une classe spécifique détermine le comportement à adopter pour le traitement des requêtes. C'est également cette classe qui va extraire des requêtes reçues un identificateur représentant le type et le contenu de cette requête. Cet identificateur sera comparé à celui extrait de la trace Ethereal afin de déterminer la correspondance entre la trame reçue et la trame attendue. Une difficulté supplémentaire, par rapport à l'émulation client, est que ces identifiants ne sont pas fournis de la même manière. En effet, dans l'émulation client les deux identifiants à comparer provenaient tous deux d'une analyse de trame Ethereal ; on pouvait utiliser la même méthode. Ici, un identifiant provient d'Ethereal, le second provient de la lecture d'une socket Java. Il faut donc bien spécifier les caractéristiques de la trame que l'on va comparer afin d'extraire les mêmes données dans les deux modes d'acquisition.

Lorsque dans le fichier de traces (A) on trouve une trame que l'on doit envoyer, il suffit de récupérer dans le fichier que tcpreplay a créé les données que cette trame contenait, et de les envoyer sur le réseau à l'aide d'une socket. Il ne faut pas oublier auparavant de déterminer sur quel port client ces données doivent être envoyées. Ce port est identifié suivant la méthode décrite dans la section 3.3.3.

Au contraire, lorsque la trame trouvée dans le fichier de traces (A) doit être attendue, et donc que cette trame est une requête, on commence par analyser les requêtes déjà reçues. Soit cette requête a déjà été reçue et on passe alors à la trame suivante dans le fichier de traces, soit cette requête est attendue jusqu'à trouver une correspondance.

Comme nous l'avons vu précédemment, un serveur peut mettre fin lui-même à une session. Il faut dans ce cas que le rejeu fasse de même. A cette fin, il faut analyser les trames contenant les fanions FIN positionnés sur 1. Soit celui-ci provient du serveur, il faut alors fermer la session concernée. Soit celui-ci provient du client et aucun traitement particulier n'est à effectuer. En effet, c'est le composant socket de la librairie Java.Net qui va s'en charger lorsque le client se déconnectera.

Fichier de traces original :

0.000000 172.20.3.162 -> 139.54.130.139 WTP+WSP WSP Connect (0x01)
0.095350 139.54.130.139 -> 172.20.3.162 WTP+WSP WSP ConnectReply (0x02)
1.020449 172.20.3.162 -> 139.54.130.139 WTP+WSP WTP Ack
1.119603 172.20.3.162 -> 139.54.130.139 WTP+WSP WSP Get (0x40)
http://nx0097.nextenso.alcatel.fr:7777/llv.wml
1.174189 139.54.130.139 -> 172.20.3.162 WTP+WSP WSP Reply (0x04): 200 OK (0x20) (WBXML 1.1,
Public ID: "-//WAPFORUM//DTD WML 1.1//EN (WML 1.1)")
3.427841 172.20.3.162 -> 139.54.130.139 WTP+WSP WTP Ack
11.562605 172.20.3.162 -> 139.54.130.139 WTP+WSP WSP Get (0x40)
http://nx0097.nextenso.alcatel.fr:7777/tools.wml
11.634577 139.54.130.139 -> 172.20.3.162 WTP+WSP WSP Reply (0x04): 200 OK (0x20) (WBXML
1.1, Public ID: "-//WAPFORUM//DTD WML 1.1//EN (WML 1.1)")
13.781267 172.20.3.162 -> 139.54.130.139 WTP+WSP WTP Ack
28.820982 172.20.3.162 -> 139.54.130.139 WTP+WSP WSP Get (0x40)
http://nx0097.nextenso.alcatel.fr:7777/ll0v.wml
28.877964 139.54.130.139 -> 172.20.3.162 WTP+WSP WSP Reply (0x04): 200 OK (0x20) (WBXML
1.1, Public ID: "-//WAPFORUM//DTD WML 1.1//EN (WML 1.1)")
31.963504 172.20.3.162 -> 139.54.130.139 WTP+WSP WTP Ack
39.062425 172.20.3.162 -> 139.54.130.139 WTP+WSP WSP Get (0x40)
http://nx0097.nextenso.alcatel.fr:7777/redir.jsp
39.276386 139.54.130.139 -> 172.20.3.162 WTP+WSP WSP Reply (0x04): 302 Moved Temporarily
(0x32)(WBXML 1.1, Public ID: "-//WAPFORUM//DTD WML 1.1//EN (WML 1.1)")
40.896146 172.20.3.162 -> 139.54.130.139 WTP+WSP WTP Ack
41.003109 172.20.3.162 -> 139.54.130.139 WTP+WSP WSP Get (0x40)
http://nx0097.nextenso.alcatel.fr:7777/ok.wml
41.130089 139.54.130.139 -> 172.20.3.162 WTP+WSP WSP Reply (0x04): 200 OK (0x20) (WBXML
1.1, Public ID: "-//WAPFORUM//DTD WML 1.1//EN (WML 1.1)")
43.002825 172.20.3.162 -> 139.54.130.139 WTP+WSP WTP Ack
54.591064 172.20.3.162 -> 139.54.130.139 WTP+WSP WSP Get (0x40) http://wap.yahoo.com/
55.138985 139.54.130.139 -> 172.20.3.162 WTP+WSP WSP Reply (0x04): 200 OK (0x20) (WBXML
1.1, Public ID: "-//WAPFORUM//DTD WML 1.1//EN (WML 1.1)")
58.376487 172.20.3.162 -> 139.54.130.139 WTP+WSP WTP Ack
58.479467 172.20.3.162 -> 139.54.130.139 WTP+WSP WSP Get (0x40)
http://wap.yahoo.com/images/yahooicon.wbmp
58.953402 139.54.130.139 -> 172.20.3.162 WTP+WSP WSP Reply (0x04): 200 OK (0x20)
(image/vnd.wap.wbmp)
61.363034 172.20.3.162 -> 139.54.130.139 WTP+WSP WTP Ack
65.006480 172.20.3.162 -> 139.54.130.139 WTP+WSP WSP Get (0x40) http://wap.yahoo.fr/
65.574397 139.54.130.139 -> 172.20.3.162 WTP+WSP WSP Reply (0x04): 302 Moved Temporarily
(0x32) (WBXML 1.1, Public ID: "-//WAPFORUM//DTD WML 1.1//EN (WML 1.1)")
67.099162 172.20.3.162 -> 139.54.130.139 WTP+WSP WTP Ack
67.183134 172.20.3.162 -> 139.54.130.139 WTP+WSP WSP Get (0x40) http://fr.wap.yahoo.com/raw?
68.442964 139.54.130.139 -> 172.20.3.162 WTP+WSP WSP Reply (0x04): 200 OK (0x20) (WBXML
1.1, Public ID: "-//WAPFORUM//DTD WML 1.1//EN (WML 1.1)")
71.777450 172.20.3.162 -> 139.54.130.139 WTP+WSP WTP Ack

.
.

Fichier de traces capturé lors du rejeu :

0.000000 192.168.123.10 -> 192.168.123.101 WTP+WSP WSP Connect (0x01)
0.096204 192.168.123.101 -> 192.168.123.10 WTP+WSP WSP ConnectReply (0x02)
1.019062 192.168.123.10 -> 192.168.123.101 WTP+WSP WTP Ack
1.117034 192.168.123.10 -> 192.168.123.101 WTP+WSP WSP Get (0x40)
http://nx0097.nextenso.alcatel.fr:7777/llv.wml
1.175021 192.168.123.101 -> 192.168.123.10 WTP+WSP WSP Reply (0x04): 200 OK (0x20) (WBXML
1.1, Public ID: "-//WAPFORUM//DTD WML 1.1//EN (WML 1.1)")
3.428697 192.168.123.10 -> 192.168.123.101 WTP+WSP WTP Ack
11.563460 192.168.123.10 -> 192.168.123.101 WTP+WSP WSP Get (0x40)
http://nx0097.nextenso.alcatel.fr:7777/tools.wml
11.634430 192.168.123.101 -> 192.168.123.10 WTP+WSP WSP Reply (0x04): 200 OK (0x20) (WBXML
1.1, Public ID: "-//WAPFORUM//DTD WML 1.1//EN (WML 1.1)")
13.782123 192.168.123.10 -> 192.168.123.101 WTP+WSP WTP Ack
28.820836 192.168.123.10 -> 192.168.123.101 WTP+WSP WSP Get (0x40)
http://nx0097.nextenso.alcatel.fr:7777/ll0v.wml
28.878812 192.168.123.101 -> 192.168.123.10 WTP+WSP WSP Reply (0x04): 200 OK (0x20) (WBXML
1.1, Public ID: "-//WAPFORUM//DTD WML 1.1//EN (WML 1.1)")
31.964358 192.168.123.10 -> 192.168.123.101 WTP+WSP WTP Ack
39.062280 192.168.123.10 -> 192.168.123.101 WTP+WSP WSP Get (0x40)
http://nx0097.nextenso.alcatel.fr:7777/redir.jsp
39.277236 192.168.123.101 -> 192.168.123.10 WTP+WSP WSP Reply (0x04): 302 Moved Temporarily
(0x32) (WBXML 1.1, Public ID: "-//WAPFORUM//DTD WML 1.1//EN (WML 1.1)")
40.897001 192.168.123.10 -> 192.168.123.101 WTP+WSP WTP Ack
41.002962 192.168.123.10 -> 192.168.123.101 WTP+WSP WSP Get (0x40)
http://nx0097.nextenso.alcatel.fr:7777/ok.wml
41.129951 192.168.123.101 -> 192.168.123.10 WTP+WSP WSP Reply (0x04): 200 OK (0x20) (WBXML
1.1, Public ID: "-//WAPFORUM//DTD WML 1.1//EN (WML 1.1)")
43.003681 192.168.123.10 -> 192.168.123.101 WTP+WSP WTP Ack
54.590918 192.168.123.10 -> 192.168.123.101 WTP+WSP WSP Get (0x40) http://wap.yahoo.com/
55.138837 192.168.123.101 -> 192.168.123.10 WTP+WSP WSP Reply (0x04): 200 OK (0x20) (WBXML
1.1, Public ID: "-//WAPFORUM//DTD WML 1.1//EN (WML 1.1)")
58.376341 192.168.123.10 -> 192.168.123.101 WTP+WSP WTP Ack
58.479309 192.168.123.10 -> 192.168.123.101 WTP+WSP WSP Get (0x40)
http://wap.yahoo.com/images/yahooicon.wbmp
58.954250 192.168.123.101 -> 192.168.123.10 WTP+WSP WSP Reply (0x04): 200 OK (0x20)
(image/vnd.wap.wbmp)
61.363888 192.168.123.10 -> 192.168.123.101 WTP+WSP WTP Ack
65.006334 192.168.123.10 -> 192.168.123.101 WTP+WSP WSP Get (0x40) http://wap.yahoo.fr/
65.574248 192.168.123.101 -> 192.168.123.10 WTP+WSP WSP Reply (0x04): 302 Moved Temporarily
(0x32) (WBXML 1.1, Public ID: "-//WAPFORUM//DTD WML 1.1//EN (WML 1.1)")
67.099016 192.168.123.10 -> 192.168.123.101 WTP+WSP WTP Ack
67.182982 192.168.123.10 -> 192.168.123.101 WTP+WSP WSP Get (0x40)
http://fr.wap.yahoo.com/raw?
68.443816 192.168.123.101 -> 192.168.123.10 WTP+WSP WSP Reply (0x04): 200 OK (0x20) (WBXML
1.1, Public ID: "-//WAPFORUM//DTD WML 1.1//EN (WML 1.1)")
71.778305 192.168.123.10 -> 192.168.123.101 WTP+WSP WTP Ack

.

3.4 Explication des limitations

Nous allons décrire dans cette section les limitations des deux solutions qui viennent d'être développées afin de pouvoir expliquer les nouvelles pistes que le chapitre suivant décrit.

Cette section est décomposée en deux sous-sections. Premièrement, nous allons exposer les limitations de l'émulation client. Ensuite, nous allons détailler les limitations de l'émulation serveur.

3.4.1. Limitations de l'émulation client

3.4.1.1. Deux machines pour rejouer

La première limitation provient de l'implémentation de tcpreplay. Afin de pouvoir réémettre les trames sur le réseau, tcpreplay utilise la librairie *libnet* et plus particulièrement la fonction *libnet_adv_write_link* qui permet d'écrire directement en se plaçant au niveau de la couche liaison de données du modèle OSI.

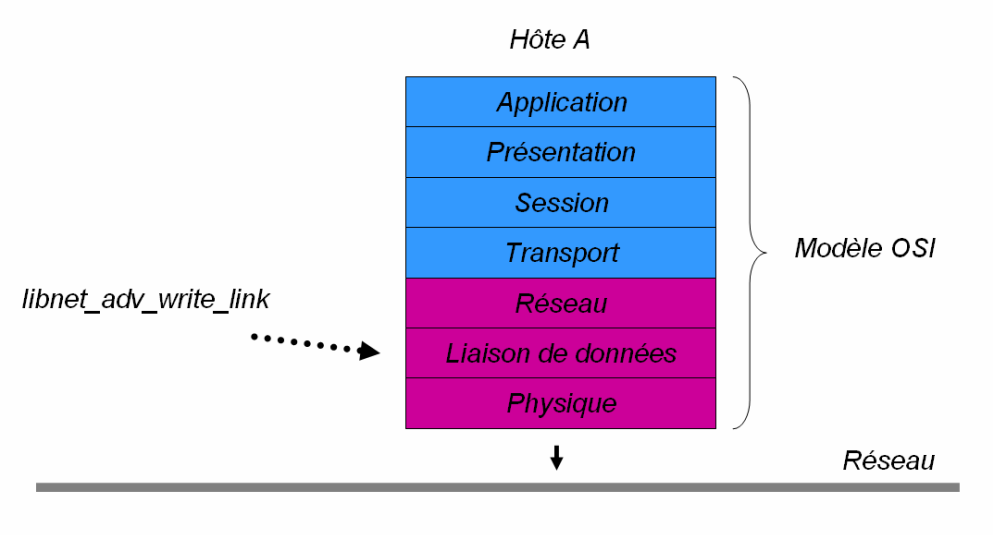

FIG 3.11 – Illustration du modèle OSI

Comme nous le voyons sur le schéma, cette fonction court-circuite la plupart des couches du modèle OSI. Cela permet donc de réémettre une trame facilement en indiquant simplement où sont les données à réémettre et la taille de ces données. En court-circuitant toutes les couches supérieures et surtout la couche liaison de données, tcpreplay n'est plus capable d'émettre sur l'interface loopback. En effet, c'est au niveau liaison de données que l'interface loopback est implémentée.

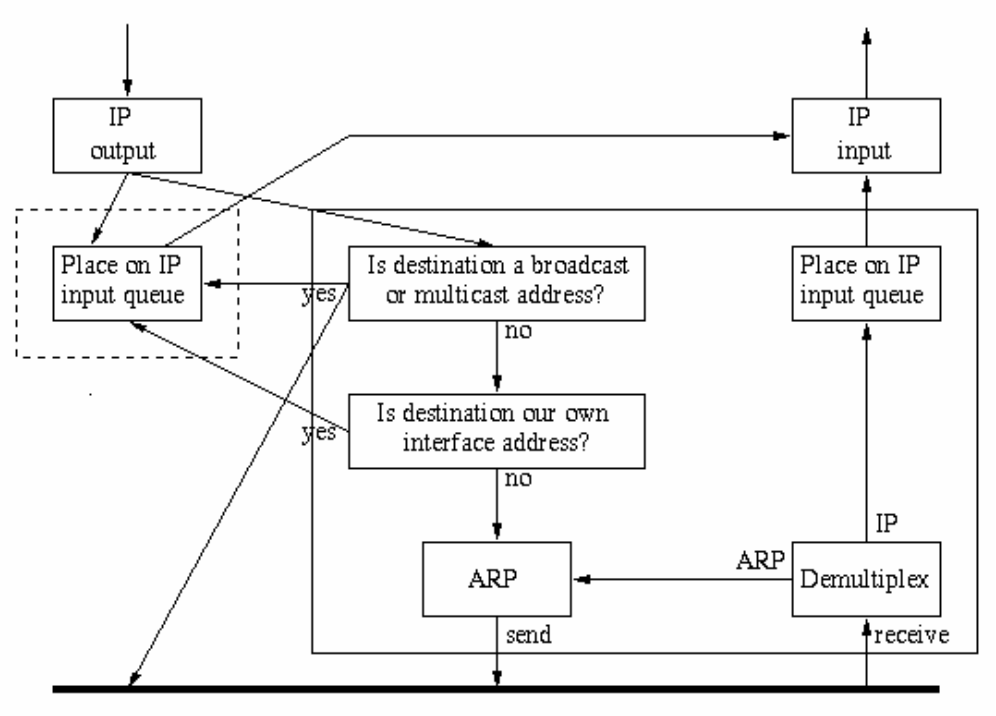

FIG 3.12 – Implémentation de l'interface loopback [Stevens 1994]

La première petite boîte encadrée en trait discontinu sur la gauche représente l'interface loopback et l'autre boîte encadrée en trait plein représente le driver Ethernet. Un datagramme IP est envoyé sur l'interface loopback si [Farmer]:

- Il est adressé directement à l'interface loopback (en général 127.0.0.1), il va donc immédiatement être mis dans la file d'attente de l'entrée IP.
- Il est destiné à une adresse broadcast ou multicast, il est recopié dans la file d'attente de l'entrée IP ainsi qu'envoyé sur le réseau
- Il est destiné à l'adresse IP de la machine.

Comme nous l'avons vu dans la figure 3.11, l'utilisation de la fonction *libnet_adv_write_link* court-circuite plusieurs couches du modèle OSI, notamment l'implémentation de l'interface loopback. De ce fait, nous sommes dans l'obligation d'utiliser deux machines distinctes pour le rejeu. La première permet l'émulation du client avec la solution développée, la deuxième étant la *Proxy Platform.*

3.4.1.2. Version de la Proxy Platform

Afin de s'assurer du bon déroulement du rejeu, il faut s'assurer que la version de la *Proxy Platform* est identique. En fait, le problème se situe au niveau de la rigueur des tests de correspondance entre trames reçues et émises. Si l'on désire pouvoir exécuter le rejeu sur différentes versions de la *Proxy Platform*, il faut alors diminuer la rigueur du test. Néanmoins, si les modifications sur le formatage des réponses sont trop importantes, notamment au niveau du volume des données échangées, le rejeu peut ne plus fonctionner correctement. En effet, si dans le rejeu la *Proxy Platform* envoie moins de données que dans le fichier de traces, il se peut que la fragmentation des messages soit différente et le rejeu va alors attendre une deuxième trame de réponse qui n'arrivera pas. Donc a priori, il faut que la *Proxy Platform* envoie au moins la même quantité de données. En effet, le problème inverse c'est – à – dire que les messages soient plus volumineux dans le rejeu que dans le fichier de traces ne posera pas de problème, la trame supplémentaire éventuelle sera simplement ignorée par l'émulateur client.

3.4.1.3. Pas de mécanisme de retransmission

Lors de l'émulation d'un client, si une trame rejouée n'arrive pas à destination pour une raison ou une autre, le rejeu risque de ne plus fonctionner correctement. En effet, il n'y aucun mécanisme permettant de vérifier le bon déroulement du rejeu à part le test de correspondance entre les trames attendues et les trames reçues. Si bien que si une des trames réémises est perdue, l'émulateur ne dispose pas de mécanisme de détection de cette perte et encore moins de mécanisme de réémission. Il faut donc que le rejeu se déroule sans accroc.

3.4.2. Limitation de l'émulation serveur

3.4.2.1. Problème de retransmission

Avant toute chose, il faut rappeler que le serveur émulé se base sur la capture de traces côté *Proxy Platform.* Ceci a de l'importance dans certains cas de figure que nous allons exposer dans cette section.

Premier cas de figure, la requête émise par la *Proxy Platform* (plus exactement sa stack http) ne parvient pas au serveur web, ou l'accusé de réception de cette requête ne revient pas à la *Proxy Platform* .

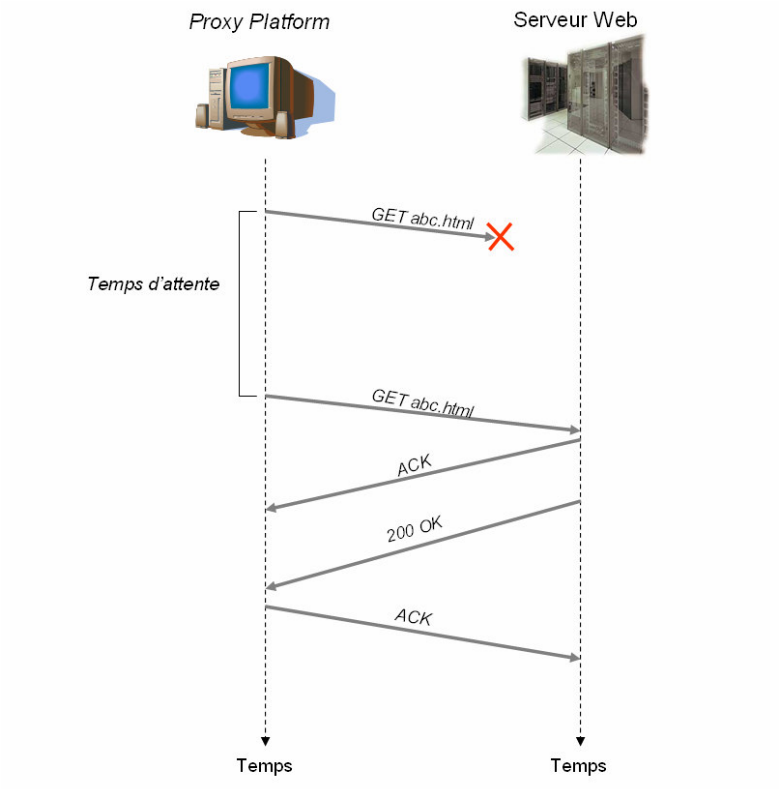

FIG 3.13 – Illustration du cas de retransmission d'une requête

Le fichier de traces contient donc les trames suivantes en omettant les trames de connexion et les trames de déconnexion :

Temps	Source	Destination	Protocole	Info
0	*Proxy Platform*	*Serveur web*	HTTP	GET abc.html HTTP/1.1
20	*Proxy Platform*	*Serveur web*	HTTP	GET abc.html HTTP/1.1
21	*Serveur web*	*Proxy Platform*	TCP	ACK Seq = xxx Ack = yyy
22	*Serveur web*	*Proxy Platform*	HTTP	HTTP/1.1 200 OK
23	*Proxy Platform*	*Serveur web*	TCP	ACK Seq = xxx Ack = yyy

Sur base de la description de la solution développée dans la section précédente, on peut s'apercevoir que le rejeu ne va pas fonctionner. En effet, nous avons expliqué que le serveur émulé utilisait une socket de la librairie Java.Net. Ce composant socket est chargé d'accepter les requêtes arrivant sur le port dont il s'occupe. Donc lorsqu'il va recevoir la première requête, celui-ci va automatiquement envoyer l'accusé de réception. De plus, il n'existe aucun moyen d'empêcher cet acquittement automatique.

Ce problème va conduire à un deadlock. En effet, le serveur émulé va procéder à l'analyse de la trame suivante et attendre une seconde fois la requête *GET abc.html* qui n'arrivera jamais puisque la première a été acquittée. La perte de l'acquittement émis par le serveur émulé conduit à une impasse identique.

Deuxième cas de figure, la réponse du serveur web n'arrive pas à destination ou l'accusé de réception de cette réponse est perdu.

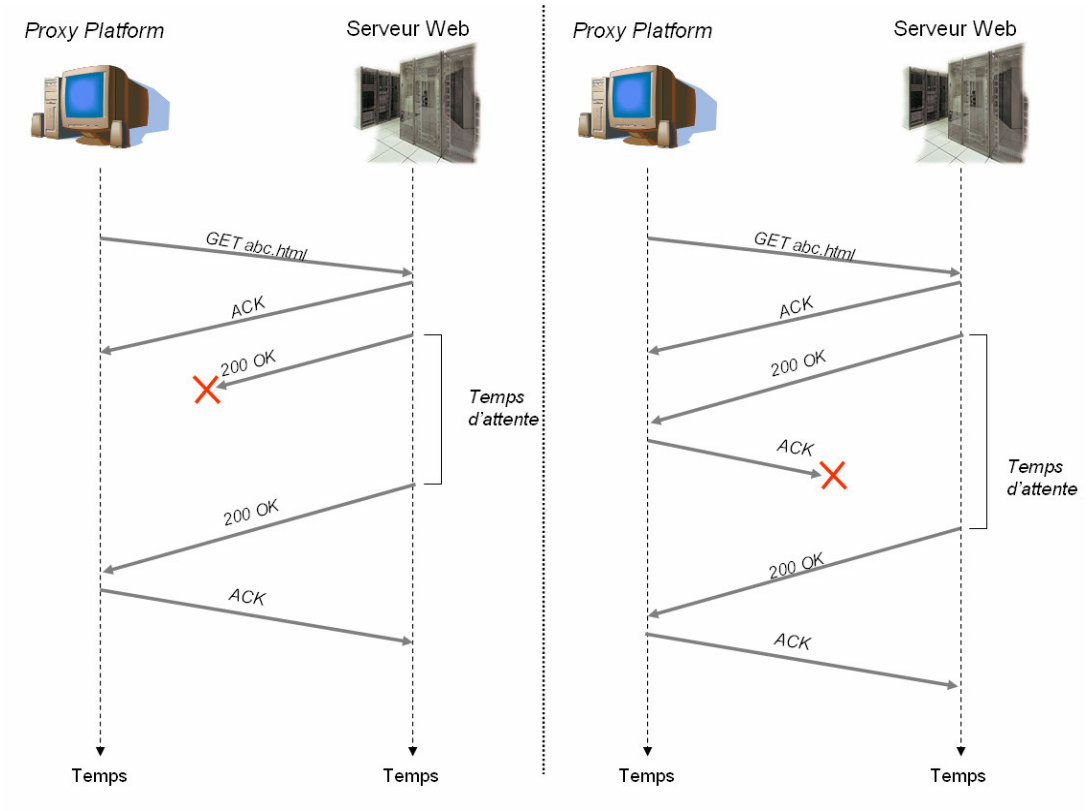

FIG 3.14 – Illustration des cas de retransmission d'une réponse

Si la réponse provenant du serveur est perdue, alors les trames contenues dans le fichier de traces seront :

Temps	Source	Destination	Protocole	Info
0	*Proxy Platform*	*Serveur web*	HTTP	GET abc.html HTTP/1.1
1	*Serveur web*	*Proxy Platform*	TCP	ACK Seq = xxx Ack = yyy
21	*Serveur web*	*Proxy Platform*	HTTP	HTTP/1.1 200 OK
22	*Proxy Platform*	*Serveur web*	TCP	ACK Seq = xxx Ack = yyy

En effet, vu que la prise de traces est effectuée côté *Proxy Platform*, on n'a pas capturé la première réponse perdue. De ce fait le rejeu ne va pas poser de problèmes ; le serveur émulé attendant vingt et une secondes entre la réception de la requête et l'émission de la réponse de manière à simuler la perte de la trame dans le scénario.

Si c'est la perte de l'accusé de réception qui entraîne la retransmission, alors le rejeu ne va pas non plus poser de problèmes. Voici le contenu du fichier de traces :

Temps	Source	Destination	Protocole	Info
0	*Proxy Platform*	*Serveur web*	HTTP	GET abc.html HTTP/1.1
1	*Serveur web*	*Proxy Platform*	TCP	ACK Seq = xxx Ack = yyy
2	*Serveur web*	*Proxy Platform*	HTTP	HTTP/1.1 200 OK
3	*Proxy Platform*	*Serveur web*	TCP	ACK Seq = xxx Ack = yyy
23	*Serveur web*	*Proxy Platform*	HTTP	HTTP/1.1 200 OK
24	*Proxy Platform*	*Serveur web*	TCP	ACK Seq = xxx Ack = yyy

On constate donc que la réponse sera émise à deux reprises comme dans le scénario original mais que la *Proxy Platform* ne prendra en compte que la première réponse, ignorant la seconde.

4. Pistes futures

4.1 Introduction

Ce chapitre va décrire les pistes envisageables afin d'améliorer les solutions développées dans le chapitre précédent. Nous allons commencer par introduire les divers outils qui existent sur Internet et montrer leurs limites.

Ensuite, nous allons décrire principalement trois pistes. La première se base sur l'existant et propose des améliorations diverses. La deuxième piste envisagera une solution apportant plus de modifications à l'existant avec des changements plus radicaux. Quant à la troisième elle envisagera une solution ne se basant plus sur l'existant tout en tenant compte de l'expérience acquise dans les solutions déjà développées.

4.1.1. Critique de l'existant

Après quelques recherches, de nouveaux outils m'ont été proposés. La première section de ce chapitre va permettre d'introduire ces outils. Nous allons commencer par une librairie Java nommée JavaSim. Ensuite nous allons nous attarder sur Click Modular Router, qui est une librairie en C++ intégrée au noyau Linux. Finalement, nous allons décrire Flowreplay (brièvement introduit au chapitre 3) qui est un programme en C permettant de rejouer des traces Ethereal mais émulant uniquement un terminal client.

4.1.2. JavaSim

JavaSim est un simulateur orienté événement développé par Hung-ying Tyan assisté de divers développeurs du département Informatique de l'université de l'Ohio (Etats-Unis). Ce

simulateur inclut déjà plusieurs modèles de protocoles utilisés sur Internet et peut être utilisé dans bon nombre de scénarios de simulation. Comme son nom l'indique, il est développé en Java et est multi- plateforme [javasim] (site officiel : http://www.j-sim.org).

En parcourant l'API de JavaSim, nous pouvons trouver une classe intéressante *TraceInput*. Cette classe permet, en se fiant aux spécifications de la documentation, la réinjection de trames contenues dans un fichier de traces.

Malheureusement, cette fonctionnalité est assez limitée. En effet, elle permet juste le rejeu d'un fichier de traces sans pratiquement aucun contrôle sur l'émission de chaque trame. Cette classe est en fait utilisée pour générer un trafic de fond simulant une activité Internet dans le but de tester le fonctionnement de différents dispositifs.

4.1.3. Click Modular Router

Click Modular Router (CMR) est un routeur logiciel modulaire développé par le MIT LCS Parallel and Distributed Operating Systems Group, le Mazu Networks, l' ICSI Center for Internet Research et l' UCLA. CMR est flexible, configurable et facile à comprendre tout en étant performant.

En fait, CMR est une interconnexion de modules appelé éléments. Ces éléments contrôlent chaque aspect du comportement d'un routeur, de la communication avec les dispositifs, de la modification de trame, du système de mise en file d'attente, de la politique de mise à l'écart de trame,…

De nouveaux éléments écrits en C++ sont faciles à intégrer et la configuration d'un routeur se fait en assemblant tous ces éléments en utilisant un langage simple.

La première fonctionnalité intéressante que l'on peut trouver dans la documentation de CMR, est la fonction *FromDump*. Elle permet la lecture de paquets dans un fichier de traces et la réécriture de ceux-ci sur le réseau en respectant le timing. Malheureusement et à l'instar de JavaSim, cette fonctionnalité permet juste le rejeu de fichier de traces dans le but de créer un trafic de fond mais n'offre pas suffisamment de contrôle sur le déroulement du rejeu.

La fonction *FromTcpdump* offre le même type de fonctionnalité excepté que le format du fichier lu est différent. En effet, il s'agit ici d'un fichier ASCII contenant la description de chaque trame capturée.

En fait, CMR offre la possibilité de lire plusieurs formats de fichiers de traces mais cette fonctionnalité est à chaque fois limitée à la création d'un trafic de fond. Il n'octroie pas suffisamment de contrôle sur le rejeu. En outre, l'utilisation de CMR contraint le développeur souhaitant l'utiliser à recompiler son noyau linux afin d'intégrer ses différents modules. Ceci pourrait constituer un frein pour des développeurs peu expérimenté en compilation de noyau linux.

4.1.4. FlowReplay

Comme évoqué au chapitre 3, tcpreplay a été développé pour rejouer des fichiers au format *libpcap* dans le but de tester des NIDS et d'autres dispositifs passifs. Néanmoins, la capacité de pouvoir se reconnecter à un serveur dans le but de tester des stacks et des applications a souvent été réclamée. Dans cette optique et afin de ne pas surcharger tcpreplay, Aaron Turner, le développeur de ces outils, a décidé de créer un nouvel outil spécifique. [Turner]

Flowreplay a été élaboré pour rejouer du trafic en se basant sur la couche réseau ou applicative et non plus en se basant uniquement sur la couche liaison de données comme le fait tcpreplay. Cela permet à flowreplay de se connecter à un ou plusieurs serveurs et permet donc de tester des applications tournant sur des serveurs réels plutôt que sur des dispositifs passifs.

Néanmoins, flowreplay n'est pas encore complètement fonctionnel car en début de développement. Toutefois, l'objectif de flowreplay est similaire à celui développé dans l'émulateur client de ce document c'est – à – dire la possibilité de rejouer un scénario afin de tester les applications tournant sur le serveur. Il serait donc intéressant de suivre son évolution.

4.2 Première solution future

La première solution envisage de reprendre le développement de l'outil décrit dans le chapitre précédent et d'apporter quelques optimisations ainsi que de régler les problèmes expliqués auparavant.

Une première amélioration sera d'afficher clairement le bon déroulement du rejeu. En effet, pour le moment, aucune information précise ne permet à l'utilisateur de savoir comment le rejeu s'est déroulé ou quels sont les problèmes rencontrés. Dans l'état, le programme affiche des informations diverses comme les connexions effectuées, les trames reçues. Mais ces informations ne sont pas claires et ne sont pas standardisées.

Il serait utile d'afficher l'état du rejeu afin de permettre à l'utilisateur de savoir ce que le programme est en train de faire, mais également, de permettre à celui-ci de voir où le rejeu a posé problème.

Ainsi par exemple, le programme pourrait afficher d'une part, la trame trouvée dans le fichier de traces, l'action que le programme a décidé d'entreprendre, ainsi que l'état de cette action. Voici un exemple de ce que le programme pourrait imprimer dans le cas d'une session WAP :

Trame trouvée	Action	Etat
WAP Connect	Sending	Sended
WAP ConnectReply	Waiting	Received
WAP Ack	Sending	Sended
WAP Get mypage.wml	Sending	Sended
WAP Reply 200 OK	Waiting	Received 200 OK
WAP Ack	Sending	Sended
WAP Disconnect	Sending	Sended

En ce qui concerne la limitation concernant l'impossibilité de retransmettre une trame perdue lors de l'émulation client, il faudrait ajouter une fonctionnalité à tcpreplay. Cette fonctionnalité permettrait d'envoyer une nouvelle fois la trame considérée comme perdue. Pour cela, il est nécessaire d'ajouter un tampon dans tcpreplay destiné à contenir la dernière trame envoyée. Lorsque le processus Java détermine par un mécanisme de timeout que le paquet a été

perdu, il demandera à tcpreplay de réenvoyer le contenu du tampon. (La demande se faisant à l'aide du message SIGUSR1[3])

Concernant le problème de retransmission de l'émulateur serveur, plusieurs solutions sont envisageables. La plus simple est évidemment de détecter la retransmission dans le fichier de traces et de ne pas en tenir compte, de passer directement à la trame suivante. Cette solution, bien que simple et efficace, n'est pas la meilleure puisque le rejeu va s'écarter du scénario original. Le fait de devoir retransmettre est un mécanisme important et la suppression de toutes les retransmissions conduit à simplifier le rejeu.

Le problème venant du manque de contrôle sur l'envoi des accusés de réception, il faut trouver un moyen de ne pas l'envoyer dans ce cas de figure. Malheureusement, il n'existe pas de socket dans une librairie Java permettant de contrôler l'envoi des accusés de réception, il serait donc nécessaire de l'implémenter nous-même. Cette solution n'est pas envisageable car elle entraîne l'implémentation de l'entièreté des mécanismes de TCP pour rajouter une seule fonctionnalité supplémentaire ; cela est démesuré.

4.3 Deuxième solution future

La deuxième solution envisage de refaire complètement l'émulateur serveur en ajoutant une fonctionnalité à tcpreplay, lui permettant de modifier la valeur de différents champs dans l'entête de chaque trame, en plus des informations déjà réécrites (port, adresse IP, adresse MAC). Rappelons qu'il est nécessaire de modifier certaines informations supplémentaires dans les entêtes d'une trame afin que celle – ci soit acceptée par le client. Ces informations sont les numéros de séquence, les numéros d'accusé de réception et éventuellement la taille de la fenêtre de réception.

Pour rappel, seule la fonctionnalité de tcpreplay permettant d'extraire les données relatives à la couche applicative d'une trame était utilisée dans l'émulateur serveur. Nous envisageons ici de nous servir de tcpreplay comme outil permettant de réinjecter les trames sur le réseau à l'instar de l'émulateur client et donc de ne plus utiliser les sockets de la librairie Java.Net.

[3] Mécanisme permettant d'avertir un processus d'un événement extérieur dans un environnement linux

L'outil tcpreplay permettant déjà de d'isoler l'entête TCP dans la structure *libnet_tcp_hdr*, il faut maintenant ajouter la fonctionnalité permettant de modifier le contenu des champs.

```
struct libnet_tcp_hdr
{
    u_int16_t th_sport;       /* source port */
    u_int16_t th_dport;       /* destination port */
    u_int32_t th_seq;         /* sequence number */
    u_int32_t th_ack;         /* acknowledgement number */
#if (LIBNET_LIL_ENDIAN)
    u_int8_t th_x2:4,         /* (unused) */
           th_off:4;        /* data offset */
#endif
#if (LIBNET_BIG_ENDIAN)
    u_int8_t th_off:4,        /* data offset */
           th_x2:4;         /* (unused) */
#endif
    u_int8_t  th_flags;       /* control flags */
    u_int16_t th_win;         /* window */
    u_int16_t th_sum;         /* checksum */
    u_int16_t th_urp;         /* urgent pointer */
};
```

Comme nous le constatons, cette structure représente exactement l'illustration de la figure 3.1. Nous pouvons donc modifier les données de l'entête à partir de cette structure qui est garnie grâce à la fonction *Get_layer4* de tcpreplay notamment utilisée pour la modification des ports.

Le premier problème à résoudre est le fait que les informations sont différentes pour chaque trame et qu'il faut donc les fournir à tcpreplay juste avant que celui-ci envoie cette trame. En effet, jusqu'à maintenant toutes les informations que tcpreplay devait réécrire étaient fournies au démarrage de tcpreplay et celui – ci utilisait les mêmes informations pour chaque trame. Il faut donc dorénavant introduire un mécanisme permettant de fournir de nouvelles informations avant l'émission de chaque trame. De plus, ces informations sont contenues dans les trames provenant du client. Il faudra donc les récupérer dans celles – ci avant de les transmettre à tcpreplay. Pour cela, nous allons utiliser une architecture similaire à l'émulateur client en apportant toutefois quelques modifications.

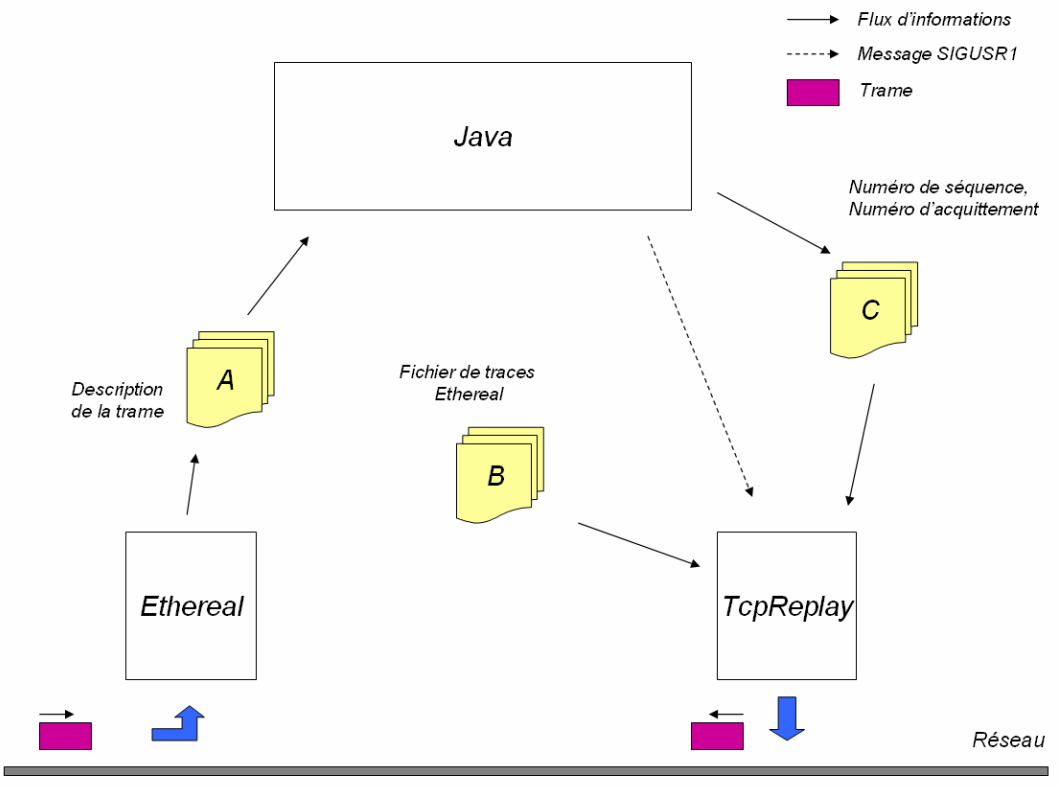

FIG 4.1 – Schéma de la solution serveur modifié

Premièrement, le rôle de chaque composant est identique à celui proposé dans la solution de l'émulateur client. Pour rappel, le processus Ethereal est utilisé pour recevoir les trames contenant les requêtes provenant du client, ces trames sont analysées par le processus Java, et les réponses adéquates sont envoyées vers ce client à l'aide de tcpreplay.

Comme nous l'avons déjà décrit dans la solution développée dans le chapitre précédent, il est nécessaire de pouvoir accepter les trames dans un ordre différent de celui du scénario. Malheureusement, les mécanismes développés pour l'émulateur client dont s'inspire maintenant l'émulation serveur, ne permettent pas de le faire. En effet, chaque trame qu'Ethereal capturait sur le réseau était envoyée au processus Java afin d'être analysée. Si celle-ci ne correspondait pas, le processus Java l'effaçait et attendait de nouveau ; si celle-ci correspondait à la trame attendue, le processus Java passait à l'analyse la trame suivante dans le fichier de trace.

Dans la nouvelle solution, il est primordial que toutes les trames reçues par Ethereal soient mémorisées afin de permettre la réception des requêtes provenant du client dans un ordre différent. Donc, chaque trame reçue par Ethereal est analysée par le processus Java et stockée dans une mémoire tampon. Lorsqu'on trouve une correspondance entre une trame attendue et

une trame stockée dans la mémoire tampon, le processus Java efface cette trame de la mémoire tampon et passe ensuite à la trame suivante dans le fichier de traces.

Nous avons vu également dans le chapitre 3 que nous avons adopté un système de parseur dédié pour analyser les trames reçues et contenues dans le fichier de traces. La particularité de ce système était de distinguer les protocoles basés sur UDP et TCP. En marge de cette séparation, cette distinction permettait également de distinguer les parseurs dédiés à l'émulation client (qui était dans notre cas basé sur UDP) des parseurs dédiés à l'émulation serveur (basé sur TCP). En effet, le mode d'acquisition des trames étant différent entre l'émulateur client (Ethereal) et l'émulateur serveur (Socket de la librairie Java.net), les méthodes implémentées dans ces parseurs étaient notablement différentes. Maintenant que le serveur utilise une architecture similaire à l'émulateur client, cette distinction n'est plus nécessaire et le système de parseurs pourrait être simplifié.

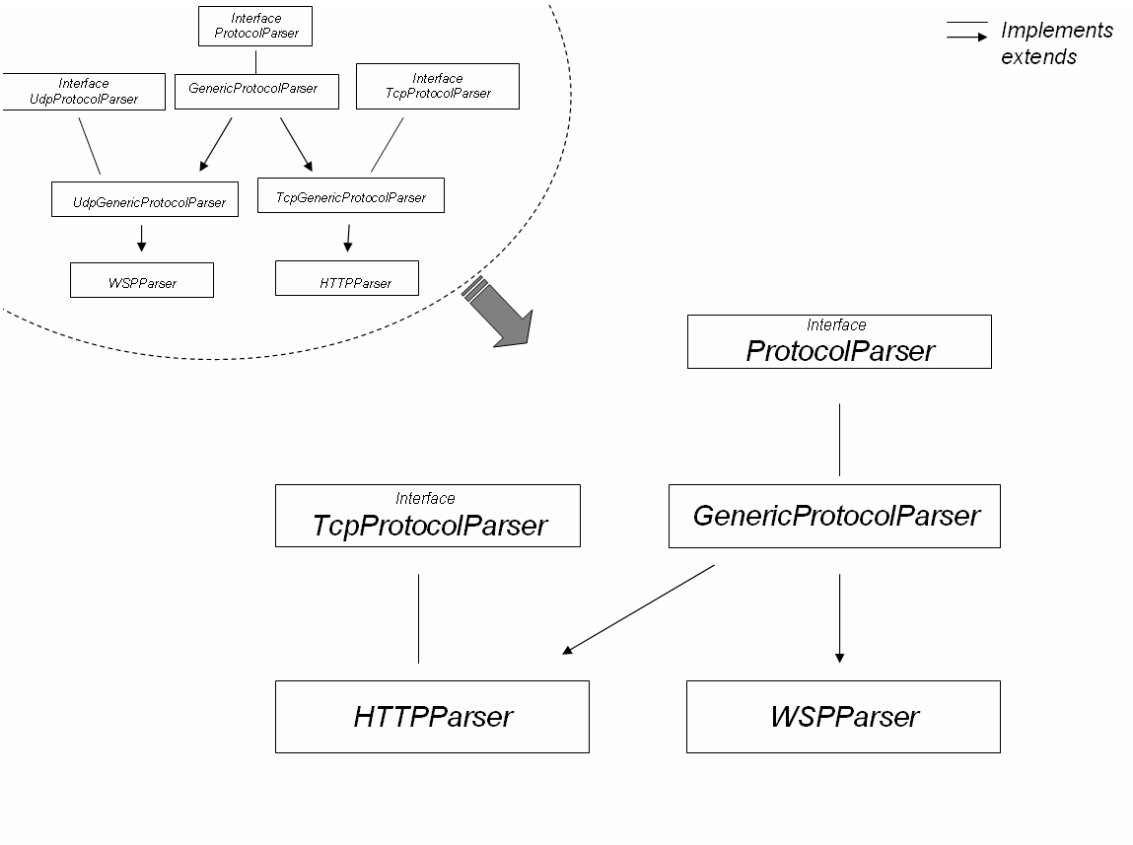

FIG 4.2 – Illustration de l'architecture modifiée du système de parseurs

Le fonctionnement reste identique, la classe GenericProtocolParser se chargeant d'extraire les informations basiques de la trame comme les adresses IP de l'expéditeur et du

destinataire, les ports utilisés ainsi que le protocole utilisé pour le niveau applicatif. Une fois ce protocole identifié, la classe GenericProtocolParser passe le relais à la classe dédiée à ce protocole afin que celle – ci puisse extraire les données pertinentes de la trame.

Par défaut, les classes dédiées doivent étendre la classe GenericProtocolParser. Toutefois, les classes dédiées à un protocole basé sur le TCP devront également étendre l'interface TcpProtocolParser afin d'obliger l'extraction des données nécessaires à tcpreplay telles que les numéros de séquence et les numéros d'accusé de réception. Rappelons que ces parseurs analysent d'une part les trames contenues dans le fichier de traces mais également les trames capturées sur le réseau lors du rejeu.

Lorsqu'un client émettra une requête HTTP vers l'émulateur serveur, Ethereal va capturer cette trame et la transmettre au processus Java. Celui-ci va ensuite analyser le contenu de cette trame grâce au système de parseurs et extraire les numéros de séquence, les numéros d'accusé de réception.

Ces informations vont permettre de calculer les numéros de séquence et les numéros d'accusé de réception de la réponse associée à la requête. Ces informations seront ensuite communiquées à tcpreplay par l'intermédiaire d'un fichier partagé afin qu'il les intègre dans la trame de réponse.

De manière similaire au problème de connexions parallèles expliqué dans le chapitre 3, il est nécessaire de conserver les numéros de séquence et les numéros d'acquittement afin de permettre de les réintégrer dans la bonne trame. En effet, nous avons vu que l'émulateur serveur devait être capable d'accepter plusieurs connexions parallèles. Dès lors, les trames peuvent arriver dans un ordre différent et il est nécessaire d'utiliser un mécanisme permettant de distinguer les différentes connexions. Le mécanisme existant, expliqué dans le chapitre 3, doit donc être modifié afin de permettre la mémorisation de ces informations complémentaires.

4.4 Troisième solution future

La troisième solution envisage l'emploi de la librairie *libnet* afin de remplacer tcpreplay et d'implémenter un outil complet. En effet, l'emploi de tcpreplay se justifiait essentiellement par le fait qu'il utilisait déjà la librairie *libnet* et que les fonctionnalités développées dans celui – ci correspondaient partiellement ou complètement aux besoins définis par le rejeu. Maintenant que nous sommes dans l'optique de modifier le comportement de tcpreplay plus en profondeur, nous allons aborder dans cette section l'éventualité de créer un outil spécifique.

4.4.1. Libnet

Nous commencerons par présenter *libnet*, pour ensuite définir son utilisation dans la résolution des traitements qui nous occupe dans ce document.

4.4.1.1. Introduction

Libnet est un API haut niveau permettant aux développeurs de construire et d'injecter des trames sur le réseau. Il fournit une interface portable et simplifiée pour créer, manipuler et injecter des trames. *Libnet* cache une grande partie de la création des paquets en se chargeant du multiplexage, de la gestion mémoire, de la gestion des entêtes, de l'alignement des bytes, des problèmes d'indépendance envers le système d'exploitation, et bien plus encore. Il permet la création de paquets aussi bien au niveau réseau qu'au niveau liaison de données. Cet API a été développé par Mike D. Schiffman assisté par une multitude d'autres développeurs. Enfin, il faut savoir que *libnet* est distribué sous licence BSD [libnet].

L'API *libnet* est réputé pour sa portabilité, sa facilité d'emploi, sa robustesse ainsi que la vitesse de correction de bogues grâce à une communauté très active. De plus, il supporte les protocoles les plus utilisés comme le montre la figure 4.3.

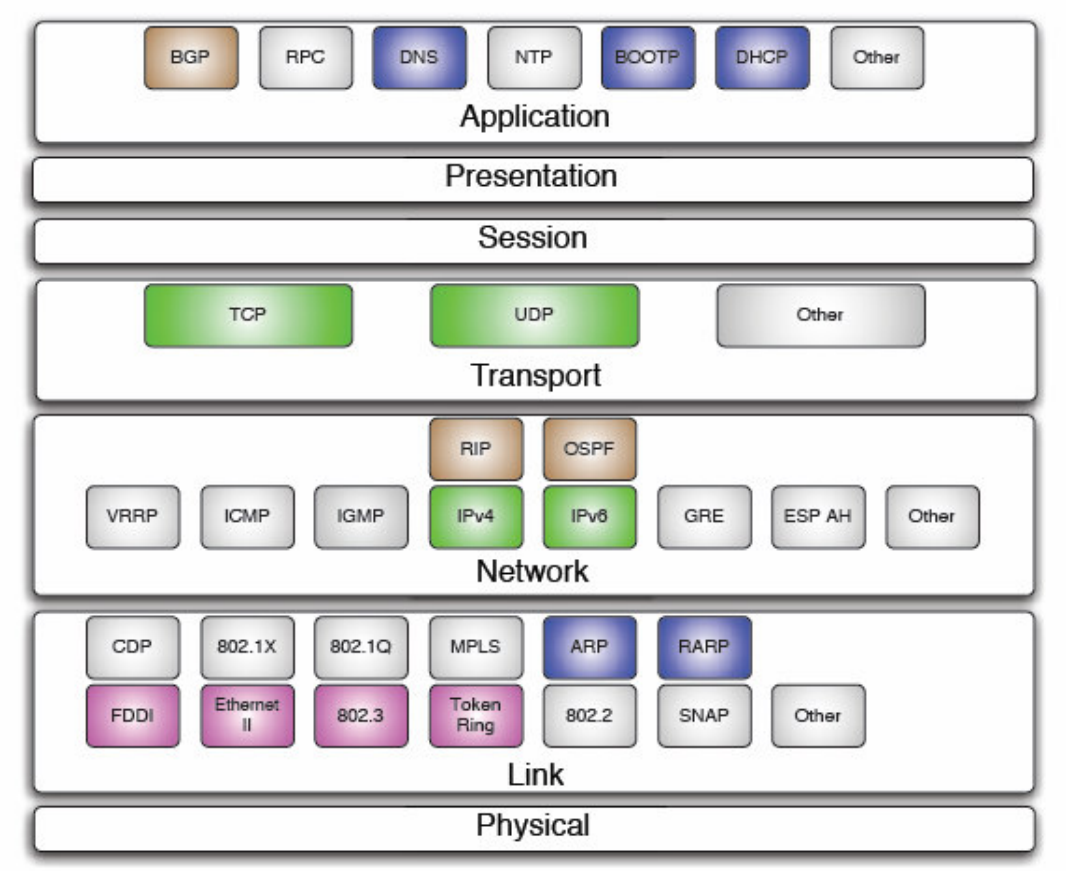

FIG 4.3 – Illustration des protocoles supportés par libnet [Schiffman]

4.4.1.2. Construire un paquet avec *libnet*

Construire un paquet avec *libnet* est une chose aisée. En effet, seul l'appel de quatre fonctions successives est nécessaire pour la création d'un paquet et de son envoi sur le réseau.

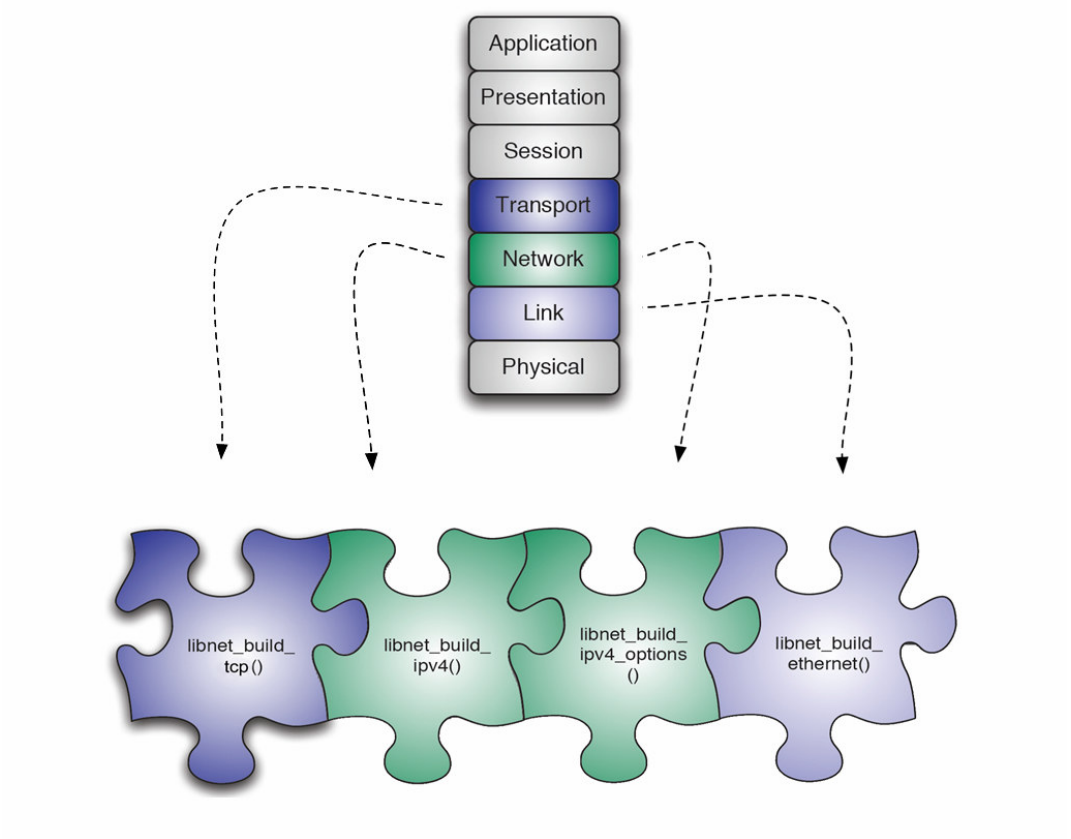

FIG 4.4 – Illustration de la construction d'un paquet TCP à l'aide de libnet [Schiffman].

Comme nous pouvons le constater sur la figure 4.4, la construction d'un paquet s'apparente à l'assemblage des pièces d'un puzzle. Chaque pièce représentant une étape de la construction du paquet, nous avons commencé dans l'exemple illustré par construire l'entête TCP. En effet, nous sommes obligés de commencer par la couche la plus haute dans le modèle OSI. Nous avons ainsi construit par la suite l'entête IPv4 pour finir par l'entête Ethernet. Le paquet étant construit et les diverses sommes de contrôle composant ce paquet étant automatiquement calculées, il ne nous reste plus qu'à l'envoyer sur le réseau. Cette dernière étape se fait par l'appel de la fonction *libnet_write()*. Afin de permettre aux lecteurs de saisir la souplesse de ce procédé, nous allons illustrer plus précisément chacune de ces fonctions et expliciter rapidement les paramètres qui les composent.

Premièrement, il est nécessaire d'initialiser le contexte *libnet* afin de pouvoir créer et injecter des paquets sur le réseau :

libnet_t* *libnet_init* (int injection_type, char* device, char* err_buf);

injection_type	LIBNET_LINK, LIBNET_RAW4
device	“fxp0”, “192.168.0.1”, NULL
err_buf	Message d’erreur si la fonction échoue

Lors de cette étape, nous spécifions le type d’injection que nous voulons utiliser parmi les deux choix possible : niveau réseau (LIBNET_RAW4) ou niveau liaison de données (LIBNET_LINK). Nous devons également spécifier le dispositif par lequel nous voulons émettre. A partir de ce moment et si la fonction ne retourne pas d’erreur, nous pouvons construire des paquets et les envoyer. En premier lieu, nous construisons l’entête TCP en utilisant la fonction *libnet_build_tcp()* décrite dans le tableau suivant :

libnet_ptag_t *libnet_build_tcp* (u_int16_t sp, u_int16_t dp, u_int32_t seq, u_int32_t ack, u_int8_t control, u_int16_t win, u_int16_t sum, u_int16_t urg, u_int16_t len, u_int8_t * payload, u_int32_t payload_s, libnet_t * l, libnet_ptag_t ptag) ;

sp,dp	Port source et port destination
seq	Numéro de séquence
ack	Numéro d’acquittement
control	Fanion de contrôle
win	Taille de la fenêtre
sum	Somme de contrôle (0 pour calcul auto)
urg	Pointeur de données urgentes
len	Taille du paquet
payload	Pointeur sur les données à transmettre
payload_s	Taille des données
l	Contexte *libnet*
ptag	Pointeur sur une entête existante afin de la modifier

Si la création de cette entête n’a pas provoqué une erreur, nous pouvons continuer en construisant l’entête IP.

libnet_ptag_t *libnet_build_ipv4* (u_int16_t len, u_int8_t tos, u_int16_t id, u_int16_t frag, u_int8_t ttl, u_int8_t prot, u_int16_t sum, u_int32_t src, u_int32_t dst, u_int8_t *payload, u_int32_t payload_s, libnet_t *l, libnet_ptag_t ptag);

len	Longueur du paquet
tos	Type de service
id	Identification
frag	Déplacement du fragment
ttl	Time To Live
prot	Protocole de couche supérieur
sum	Somme de contrôle
src	Adresse IP source
dst	Adresse IP destinataire
payload	Pointeur sur les données à transmettre
payload_s	Taille des données
l	Contexte *libnet*
ptag	Pointeur sur une entête existante afin de la modifier

Il nous reste alors à construire l'entête ethernet afin que ce paquet puisse être envoyé sur le réseau :

libnet_ptag_t *libnet_build_ethernet* (u_int8_t* dst, u_int8_t* src, u_int16_t type, u_int8_t *payload, u_int32_t payload_s, libnet_t* l, libnet_ptag_t ptag);

dst	Adresse ethernet de destination
src	Adresse ethernet de source
type	Protocole de la couche supérieur
payload	Pointeur sur les données à transmettre
payload_s	Taille des données
l	Contexte *libnet*
ptag	Pointeur sur une entête existante afin de la modifier

Le paquet est désormais complet et peut-être envoyé sur le réseau. Pour cela, il suffit d'appeler la fonction *libnet_write (libnet_t* l)* avec en paramètre le contexte *libnet* que nous utilisons depuis la création du paquet.

4.4.2. Description de la solution utilisant *libnet*

Avant toute chose, l'utilisation du logiciel Ethereal ou de sa version console n'est pas remise en cause dans cette solution. Certes, nous pourrions construire un outil spécifique basé sur la librairie *libpcap* permettant de capturer et d'analyser des trames provenant du réseau. Cependant, les fonctionnalités qu'Ethereal propose sont complètes et correspondent totalement aux besoins que nous avons. Il est inutile d'envisager son remplacement par un outil spécifique.

La toute première chose est d'extraire du fichier de traces les données de la couche applicative. Deux solutions sont possibles, soit l'utilisation de cette fonctionnalité présente dans tcpreplay, soit l'utilisation de la librairie *libpcap* (et en particulier de la méthode pcap_next). Ces deux solutions sont valables, néanmoins, vu que tcpreplay ne pose aucun problème quant à cette fonctionnalité, nous ne voyons aucune raison particulière de réécrire cette fonctionnalité

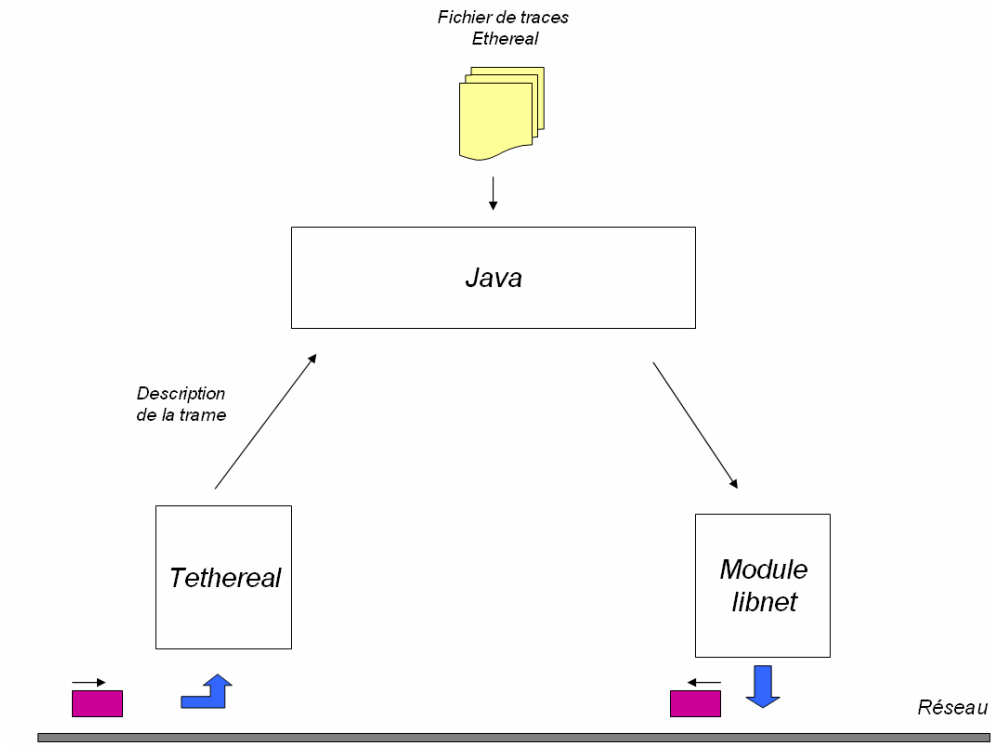

FIG 4.5 – Illustration de la solution utilisant libnet

En ce qui concerne la capture des trames provenant du client, le système reste identique avec l'emploi de tethereal. Celui-ci communique toutes les trames au processus Java afin que celui-ci les analyse. Lors de l'envoi d'une trame, le processus Java prépare la trame à envoyer

en rassemblant les données que celle-ci doit contenir (extraites du fichier de traces), mais en calculant également toutes les données contenues dans les entêtes des divers protocoles à utiliser. Par exemple, pour envoyer une trame HTTP, le processus Java va réunir toutes les informations contenues dans le tableau suivant en considérant que la trame analysée dans le fichier de traces est la réponse HTTP à envoyer :

Information provenant de la trame émise par le client	Information provenant du fichier de traces
Adresse Ethernet destination	
Adresse Ethernet source	
Type de protocole de couche supérieur	Type de protocole de couche supérieur
	Type de service
	Taille du datagramme
	Identification
	Déplacement du fragment
	TTL
	Protocole de couche supérieur
Adresse IP source	
Adresse IP destination	
Port source	
Port destination	
Numéro de séquence (calcul)	
Numéro d'accusé de réception (calcul)	
	Fanion de contrôle
	Fenêtre de réception
	Pointeur de données urgentes
	Données transmises

Comme nous pouvons le constater, certaines informations que l'on devait au préalable fournir explicitement à tcpreplay sont ici récupérées automatiquement (les différentes adresses IP et Ethernet ainsi que les ports utilisés). Les autres informations sont soit calculées, soit récupérées dans le fichier de traces.

Une fois toutes ces informations récoltées par le processus Java, nous devons les transmettre au module *libnet*. Plusieurs solutions sont possibles quant à l'implémentation de ce module : la première consiste à simplement transmettre toutes ces informations par ligne de

commande ; les spécifications du module *libnet* étant alors de récupérer ces informations, d'ouvrir un contexte *libnet*, de construire le paquet et de l'envoyer sur le réseau. Cette solution, simple et fonctionnelle, comporte néanmoins au moins un défaut. En effet, chaque émission de trame provoquera la création d'un processus, ce qui nécessite du temps et des ressources machines. Toutefois, vu que les performances ne sont pas une des spécifications principales du rejeu, cette solution reste envisageable.

Une seconde solution serait d'implémenter une interface JNI vers les différentes fonctions nécessaires au rejeu. De cette manière, les fonctionnalités de la librairie *libnet* seraient accessibles comme si elle était programmée en Java. L'avantage de cette méthode est de simplifier la programmation ultérieure. En effet, une fois l'interface JNI vers *libnet* construite et fonctionnelle, l'accès à ses fonctionnalités se fera de manière complètement transparente pour le développeur (voir annexe A).

Finalement, le développement d'une interface JNI vers les fonctionnalités de *libnet* est certes assez fastidieuse mais elle facilitera l'envoi de trames sur le réseau tout en contrôlant tout les paramètres composant cette trame.

Conclusion

Nous avons tenté de sensibiliser le lecteur à la problématique et aux enjeux spécifiques du rejeu de traces. Nous avons montré à ce dernier les avantages qu'un outil de ce genre permet d'apporter dans la réalisation de logiciels distribués. Pour rappel, ces avantages sont principalement de deux ordres : temporel en permettant d'automatiser une partie des tests, qualitatif en permettant l'augmentation de la qualité du produit.

Au cours d'un stage dans la société *Nextenso SA,* nous avons été amené à réaliser un logiciel permettant de rejouer facilement des fichiers de traces Ethereal dans le but de tester le bon fonctionnement des couples Stack – Agent de la *Nextenso Proxy Platform.*

Nous avons développé un outil répondant au cahier des charges défini en début de stage et en avons montré les limitations. Nous avons ensuite ouvert dans le dernier chapitre de nouvelles pistes permettant de remédier à ces limitations.

En ce qui concerne l'état de l'art du rejeu de traces, les différentes recherches n'ont pas fourni énormément de documentation à ce sujet. Néanmoins, au fur et à mesure de cette année, les références n'ont cessé d'augmenter, preuve que ce type d'outil est de plus en plus demandé. Un premier outil, flowreplay, est disponible sur Internet, bien qu'à un stade de développement précoce, mais dont l'aboutissement sera certainement très intéressant. Nous avons également remarqué que plusieurs librairies ont été créées de façon à permettre à des développeurs d'implémenter plus facilement ce genre d'outil, en particulier la libraire *libnet.*

Ce type de logiciel de rejeu sera de plus en plus intéressant car les systèmes distribués sont de plus en plus répandus. En effet, bien que les performances des machines ne cessent d'augmenter, dans le même temps la complexité des traitements qu'elles doivent exécuter, combiné à des temps de réponse de plus en plus faible font de l'architecture système distribué une solution fréquemment utilisée. Elle apporte une capacité de traitement facilement adaptable aux besoins fonctionnels ainsi qu'aux moyens financiers d'une société cliente, une souplesse

d’utilisation inégalée et une robustesse à toute épreuve. Cette architecture s’accompagne malheureusement d’une complexité de développement rapidement croissante ainsi que d’une maintenance plus ardue.

Bibliographie

[Kurose&Ross] James Kurose et Keith Ross, *Analyse structurée des réseaux (seconde édition),*Pearson Education, Paris, France, 2003.

[Stevens 1994] Richard Stevens, *TCP/IP Illustrated, Volume 1*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1994.

[Farmer] Mick Farmer, *Loopback Interface*, http://www.dcs.bbk.ac.uk/~mick/ academic/network s/msc/data-link/loopback.shtml#stevens1994, non daté, date de consultation:28/04/2005.

[libnet] Mike Schiffman, *The Libnet Packet Construction Library*, http://www.packetfactory.net/Projects/Libnet/, 1/03/2004, date de consultation: 15/05/2005.

[Schiffman] Mike Schiffman, *The Evolution of Libnet*, The 13th RSA Conference San Francisco, February 2004.

[javasim] Bruno Quoitin, *The JavaSim simulator and some extensions*, http://www.info.ucl.ac.be/~bqu/jsim/, non daté, date de consultation: 05/05/2005.

[CMR] Eddie Kohler, Benjie Chen, Doug De Couto, Frans Kaashoek, Robert Morris, Max Poletto, *The Click Modular Router Project*, http://pdos.csail.mit.edu/click/, non daté, date de consultation: 05/05/2005.

[Turner] Aaron Turner, *Tcpreplay: Pcap editing and replay tools for *NIX*, http://tcpreplay.sourceforge.net/, non daté, date de consultation: 01/04/2005.

[Pillou] Jean-François Pillou*, Introduction à la technologie WAP*, http://www.commentcamarche.net/wap/wapintro.php3, non daté, date de consultation : 28/03/2005.

[Marquet] Renaud Marquet, O*ptimisation d'un logiciel de transmission de MMS*, mémoire de fin d'étude, FUNDP Namur Belgique, Juin 2004.

Annexe

Annexe A

Interface JNI vers la librairie libnet.

Pour l'envoi d'une trame HTTP, il faut donc permettre l'accès à quatre fonctions. Nous allons définir ces méthodes en Java suivant les normes JNI :

```
public class Jlibnet {

        public native void libnet_init (int injection_type, String device, String err_buf);

        public native int libnet_build_tcp (int sp, int dp, int seq, int ack, int control, int win, int sum,
                                                int  urg, int len, String filename, int ptag);

        public native int libnet_build_ipv4 (int len, int tos, int id, int frag, int ttl, int prot, int sum,
                                                int src, int dst, String filename, int ptag);

        public native int libnet_build_ethernet (String dst, String src, int type, String filename,
                                                int ptag);

        static {
                System.loadLibrary("JLibnet");
        }
        public Jlibnet (int injection_type, String device, String err_buf){
                this.libnet_init(injection_type,device,err_buf);
        }
}
```

Après exécution de la commande *javah –jni JLibnet*, nous disposons d'un fichier d'entête pour l'interface en C à développer :

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class Jlibnet */

#ifndef _Included_Jlibnet
#define _Included_Jlibnet
#ifdef __cplusplus
extern "C" {
```

```
#endif
/*
 * Class:     Jlibnet
 * Method:    libnet_init
 * Signature: (ILjava/lang/String;Ljava/lang/String;)V
 */
JNIEXPORT void JNICALL Java_Jlibnet_libnet_1init
  (JNIEnv *, jobject, jint, jstring, jstring);


/*
 * Class:     Jlibnet
 * Method:    libnet_build_tcp
 * Signature: (IIIIIIIIILjava/lang/String;I)I
 */
JNIEXPORT jint JNICALL Java_Jlibnet_libnet_1build_1tcp
  (JNIEnv *, jobject, jint, jint, jint, jint, jint, jint, jint, jint, jint, jstring, jint);


/*
 * Class:     Jlibnet
 * Method:    libnet_build_ipv4
 * Signature: (IIIIIIIILjava/lang/String;I)I
 */
JNIEXPORT jint JNICALL Java_Jlibnet_libnet_1build_1ipv4
  (JNIEnv *, jobject, jint, jint, jint, jint, jint, jint, jint, jint, jint, jstring, jint);


/*
 * Class:     Jlibnet
 * Method:    libnet_build_ethernet
 * Signature: (Ljava/lang/String;Ljava/lang/String;ILjava/lang/String;I)I
 */
JNIEXPORT jint JNICALL Java_Jlibnet_libnet_1build_1ethernet
  (JNIEnv *, jobject, jstring, jstring, jint, jstring, jint);

#ifdef __cplusplus
}
#endif
#endif
```

A partir de cette entête, il nous reste à construire les fonctions en C faisant appel aux fonctions de la librairie *libnet*. Evidemment, il faut avant tout convertir les entiers passés en arguments, qui sont tous sur 32 bits, dans le type demandé par les fonctions de la librairie *libnet* (cfr les fonctions décrite à la section 4.4.1.2). En ce qui concerne l'acquisition des données à transmettre, nous avons choisi de transmettre au module en C l'emplacement du fichier les contenant afin que celui-ci puisse le lire plutôt que de passer directement les octets composant cette réponse.